Implementación del algoritmo RX para la detección de anomalías en imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable

Carlos Colomé García
Gonzalo Pericacho Sánchez

PROYECTO DE SISTEMAS INFORMÁTICOS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE CARRERA

Madrid, junio de 2013

Director: Daniel Mozos Muñoz
Colaborador: Carlos González Calvo

Autorización

Los ponentes Gonzalo Pericacho Sánchez, con DNI 05208613X y Carlos Colomé García, con DNI 11838806Q, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria como el código.

Gonzalo Pericacho Sánchez

Carlos Colomé García

En Madrid, junio de 2013

Agradecimientos

En primer lugar quisiéramos agradecer a Daniel Mozos Muñoz la oportunidad que nos ha brindado para realizar este proyecto y su apoyo durante el transcurso del mismo. A Carlos González Calvo, por su constante disponibilidad para resolvernos dudas, asesorarnos y guiarnos en el desarrollo del proyecto.

También agradecer a todos nuestros compañeros y amigos de la universidad, sobre todo a Sergio y Concepción por las largas horas dedicadas al estudio durante el transcurso de la carrera y el mutuo apoyo ofrecido para poder sacarla adelante.

A nuestros familiares, por su cariño inestimable y por concedernos la oportunidad de poder estudiar la carrera que nos gusta y motiva.

Y finalmente, no podemos dejar de agradecer la gran paciencia de Helena y Alba por aguantarnos en los malos momentos y animarnos cuando la moral estaba baja.

A todos ellos, muchas gracias.

Índice

Página
Índice de figuras....IX
Índice de tablas....XI
Resumen....XIII
Abstract....XV
Capítulo 1. Motivaciones y objetivos....1
1.1. Aportaciones del proyecto....2
Capítulo 2. Análisis hiperespectral....3
2.1. Concepto de imagen hiperespectral....3
2.2. Sensores hiperespectrales....6
2.3. Ejemplos de sensores hiperespectrales....8
2.4. Utilidad de imágenes hiperespectrales....9
Capítulo 3. Hardware reconfigurable....11
3.1. FPGAs como tecnología reconfigurable....12
3.2. Tipos de configuración de las FPGAs....14
3.3. Diseño del hardware para FPGAs....16
3.4. Ventajas de las FPGAs....17
3.5. Inconvenientes de las FPGAs....18
3.6. FPGAs en misiones de observación remota....18
Capítulo 4. Detección de anomalías....21
4.1. Algoritmo RX....22
Capítulo 5. Implementación en FPGA del algoritmo RX....23
5.1. Visión general de la implementación....23
5.2. Implementación algoritmo RX....25
5.2.1. Estructura de las memorias....27
5.2.2. Módulo matriz inversa....29
5.2.2.1. Método de Gauss....31
5.2.2.2. Ruta de datos....34
5.2.2.3. Controlador de memoria....38
5.2.2.4. Unidad de control....41

5.2.3. Módulo multiplicador matricial 49
5.2.3.1. Ruta de datos 51
5.2.3.2. Banco de registros de desplazamiento 54
5.2.3.3. Unidad de control 55
5.2.4. Modulo lista ordenada 59
5.2.5. Unidad de control RX 62
Capítulo 6. Resultados experimentales 67
6.1. Plataforma reconfigurable 67
6.2. Conjunto de imágenes hiperespectrales 68
6.2.1. AVIRIS WTC 68
6.2.2. HYDICE 70
6.3. Evaluación de las anomalías detectadas 72
6.4. Evaluación del rendimiento 77
Capítulo 7. Conclusiones 81
Bibliografía 83
Acrónimos 91

Índice de figuras

2.1. Concepto de imagen hiperespectral 4
2.2. Proceso de toma de imágenes hiperespectrales por el sensor AVIRIS 5
2.3. Clases de píxeles y sus firmas hiperespectrales 5
2.4. Diagrama del proceso de espectrometría de imágenes con sus elementos básicos 6
2.5. Curvas de reflectancia espectral para distintos materiales 7
3.1. Comparación de flexibilidad y rendimiento de las distintas opciones de computación 12
3.2. Diseño interior de una FPGA 13
3.3. Diagramas de los distintos modos de configuración de la FPGA 14
3.4. Modelos de configuración dinámica 16
5.1. Esquema general del módulo RX 24
5.2. Notación en punto flotante 25
5.3. Interconexión entre los submódulos del módulo RX, sus puertos y señales 26
5.4. Estructura interna de las memorias utilizadas 28
5.5. Interconexión entre los módulos que componen el módulo matriz inversa 30
5.6. Ejemplo de ruta de datos correspondiente al módulo matriz inversa para un tamaño de matriz 4x4 38
5.7. Diagrama de estados del controlador de memoria 39
5.8. Diagrama de estados del módulo inversa 48
5.9. Esquema general del módulo del multiplicador matricial 50
5.10. Ejemplo de estructura e interconexión de los componentes pertenecientes a la ruta de datos del multiplicador matricial para un tamaño de matriz K 4x4 53
5.11. Diagrama de estados de la unidad de control del multiplicador matricial 57
5.12. Ejemplo de cableado del módulo lista ordenada. En este caso para almacenar 4 anomalías 61
5.13. Diagrama de estados de la unidad de control del módulo RX 66
6.1. Composición en falso color de la imagen hiperespectral AVIRIS obtenida sobre la zona del WTC en la ciudad de Nueva York, cinco días después del

atentado terrorista del 11 de septiembre de 2001. El recuadro en rojo marca la zona donde se sitúa el WTC en la imagen 70

6.2. (A) Representación en falso color de la escena hiperespectral HYDICE y (B) mapa verdad-terreno para la escena..... 71

6.3. Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen AVIRIS WTC 72

6.4. Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen HYDICE..... 73

6.5. (A) Representación en falso color de la escena hiperespectral WTC y (B) su información de realidad sobre el terreno asociado 74

6.6. (A) Píxeles anómalos detectados por la implementación propuesta en el total de la imagen AVIRIS WTC y (B) en la zona del WTC..... 76

6.7. Píxeles anómalos detectados por la implementación propuesta en el total de la imagen HYDICE..... 77

Índice de tablas

Tabla 5.1: Tabla de verdad de multiplexores lista ordenada. 60

Tabla 5.2: Tabla de verdad de módulo asignado para lecturas o escrituras lista ordenada 63

Tabla 6.1 : propiedades de la FPGA Virtex-5QV XQR5VFX130 68

Tabla 6.2: Características de la imagen hiperespectral AVIRIS obtenida sobre la zona del World Trade Center en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de septiembre de 2001 69

Tabla 6.3: Características de la imagen hiperespectral HYDICE compuesta por 15 paneles puestos sobre la escena. 71

Tabla 6.4. Propiedades de los puntos calientes etiquetados en la Figura 6.7(a) 75

Tabla 6.5: Resumen de los recursos utilizados para la implementación del algoritmo RX en la FPGA Virtex-5QV XC5VFX130T. 78

Tabla 6.6: Resumen de tiempos y frecuencia de la implementación del algoritmo RX en la FPGA Virtex-5QV XC5VFX130T. 78

Tabla 6.7: Comparación de las distintas implementaciones del algoritmo RX 79

Resumen

En los últimos años, el análisis de imágenes hiperespectrales en observación remota, se ha convertido en un campo de investigación muy activo. Actualmente, debido a la resolución espacial disponible en los sensores presentes en plataformas aéreas o satelitales y a la manera en que aparecen los distintos materiales en la naturaleza, en una imagen hiperespectral podemos encontrar dos tipos de píxeles: puros y mezclas. La diferencia entre ellos es que su firma espectral se corresponde con un único material (puro) o es la combinación de varios de ellos (mezcla).

Para realizar el análisis de la composición de cada uno de los píxeles de la imagen hiperespectral se utiliza frecuentemente la técnica de *desmezclado espectral*. En determinadas situaciones no nos interesa realizar un análisis completo de la imagen, generalmente por el excesivo tiempo de ejecución que ello conllevaría, y es suficiente aplicar técnicas de análisis consistentes en la detección de anomalías. Para realizar dicha detección, existe un enfoque desarrollado por Reed y Yu, el cual es conocido comúnmente como *algoritmo RX*. Es en esta última técnica en la que se basa este trabajo.

En este proyecto fin de carrera se ha diseñado e implementado el *algoritmo RX* completo a excepción de la obtención de la matriz de covarianza, la cual ya se supondrá calculada. Para la especificación de dicha implementación se ha empleado el lenguaje de descripción hardware VHDL y se ha orientado a su uso en plataformas de hardware reconfigurable del tipo Field Programmable Gate Array (FPGA).

Palabras clave: Algoritmo RX, Detección de anomalías, FPGA, Hardware reconfigurable, Imagen hiperespectral, VHDL.

Abstract

In the latest years, remotely sensed hyperspectral imaging has become a very active research field. Nowadays, due to the available spatial resolution of the sensors in remote sensing of the Earth and the way the different materials appear in nature, in a hyperspectral image can be found two different types of pixels: pures and mixtures. The difference between them is that its spectral sign corresponds to a unique material (pure ones) or is the blend of some of them (mixture ones).

To perform the analysis of each pixel's composition of the hyperspectral image it is frequently used the technique of *spectral unmixing*. In certain situations, it is not of interest to perform a whole analysis of the image, usually due to the excessive runtime that will take, and is enough with only applying techniques of anomaly detection. A well-known approach for anomaly detection was developed by Reed and Yu, and is referred to as the RX algorithm. It is in this technique in what it is based this work.

In this project it has been designed and implemented the whole RX algorithm but the covariance matrix, which will be assumed as given. The design proposed has been developed in reconfigurable platforms like Field Programmable Gate Arrays (FPGAs) using the VHDL language for its specification.

Keywords: Anomaly detection, FPGA, Hyperspectral image, Reconfigurable hardware, RX algorithm, VHDL.

Capítulo 1

Motivaciones y objetivos

En los últimos años, el análisis de imágenes hiperespectrales en observación remota, se ha convertido en un campo de investigación muy activo. Actualmente, debido a la resolución espacial disponible en los sensores presentes en plataformas aéreas o satelitales y a la manera en que aparecen los distintos materiales en la naturaleza, en una imagen hiperespectral podemos encontrar dos tipos de píxeles: puros y mezclas. La diferencia entre ellos es que su firma espectral se corresponde con un único material (puro) o es la combinación de varios de ellos (mezcla).

Para realizar el análisis de la composición de cada uno de los píxeles de la imagen hiperespectral se utiliza frecuentemente la técnica de *desmezclado espectral*. En determinadas situaciones no nos interesa realizar un análisis completo de la imagen, generalmente por el excesivo tiempo de ejecución que ello conllevaría, y es suficiente aplicar técnicas de análisis consistentes en la detección de anomalías. Para realizar dicha detección, existe un enfoque desarrollado por Reed y Yu, el cual es conocido comúnmente como *algoritmo RX*. Es en esta última técnica en la que se basa este trabajo.

En este proyecto fin de carrera se ha diseñado e implementado el *algoritmo RX* completo a excepción de la obtención de la matriz de covarianza, la cual ya se supondrá calculada. Para la especificación de dicha implementación se ha empleado el lenguaje de descripción hardware VHDL y se ha orientado a su uso en plataformas de hardware reconfigurable del tipo Field Programmable Gate Array (FPGA).

1.1. Aportaciones del proyecto

Este proyecto presenta una posible implementación hardware del algoritmo RX de Reed y Yu para la detección de anomalías en imágenes hiperespectrales. La especificación se ha realizado mediante el lenguaje VHDL para su posterior implementación sobre FPGA. La elección de las FPGAs como plataforma para el desarrollo se ha basado en las ventajas e inconvenientes que se expondrán a lo largo de esta memoria.

Capítulo 2

Análisis hiperespectral

En este apartado se van a explicar qué son las imágenes hiperespectrales y los principales conceptos relacionados con el estudio y análisis hiperespectral.

2.1. Concepto de imagen hiperespectral

Los actuales sensores hiperespectrales toman imágenes digitales en una gran cantidad de canales (longitudes de onda o bandas) muy próximos entre sí. De esta forma obtenemos para cada píxel de la imagen una firma espectral. Dicha firma espectral puede ser representada como una gráfica, la cual es característica de cada material [1].

Al acabar de tomar las muestras de todos los píxeles de la imagen y teniendo en cuenta que para cada uno de ellos se ha obtenido su firma espectral, obtenemos una estructura de datos en forma de cubo [2]. Dicho cubo está formado por tres dimensiones. Dos de ellas determinan la ubicación espacial de un píxel en la imagen (a estas dos dimensiones normalmente se las denomina *líneas* y *muestras*) y la tercera contiene la reflectancia en las diferentes longitudes de onda de cada píxel o lo que es lo mismo, la firma espectral de cada píxel. En la *figura 2.1* se muestra en detalle lo explicado anteriormente.

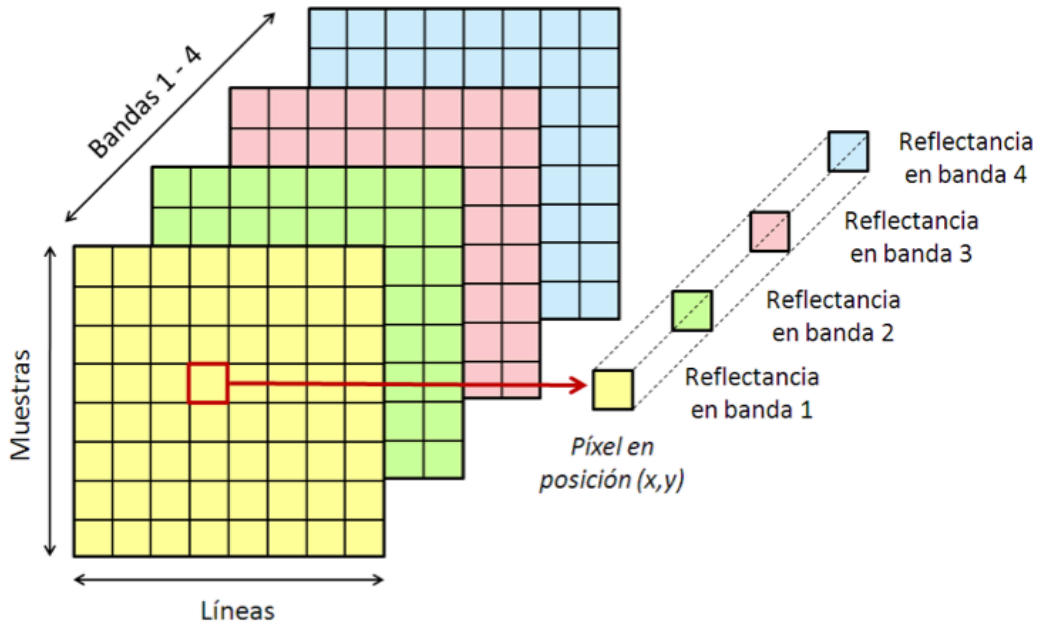


Figura 2.1: Concepto de imagen hiperespectral.

La obtención de los valores de longitud de onda por parte del sensor se realiza midiendo la reflectancia que provocan los materiales de la imagen en las distintas longitudes de onda [3]. Cuantas más bandas pueda captar el sensor por cada píxel y menos separación haya entre ellas, más detallada quedará la firma espectral del mismo.

El procedimiento de captura de imágenes hiperespectrales queda reflejado en la *figura 2.2*. En el diagrama se ha considerado como ejemplo el sensor AVIRIS (*Airborne Visible Infra-Red Imagine Spectrometer*). Dicho sensor dispone de un rango de longitudes de onda entre 0.4 y 2.5 μm utilizando 224 canales y una resolución de aproximadamente 10 nm [4].

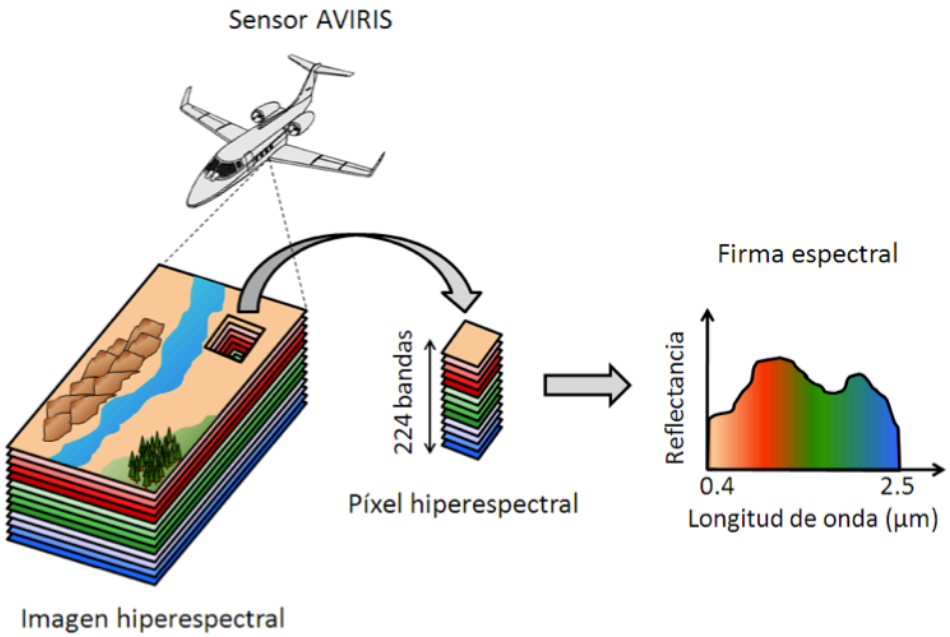


Figura 2.2: Proceso de toma de imágenes hiperespectrales por el sensor AVIRIS.

Debido a la distribución natural de los materiales en la superficie de la Tierra, al tomar una muestra con un sensor hiperespectral podemos distinguir dos tipos de píxeles en la imagen: píxeles mezcla y píxeles puros. Los píxeles mezcla son aquellos en los que coexisten distintos materiales, mientras que en los píxeles puros se distingue claramente un único material [5].

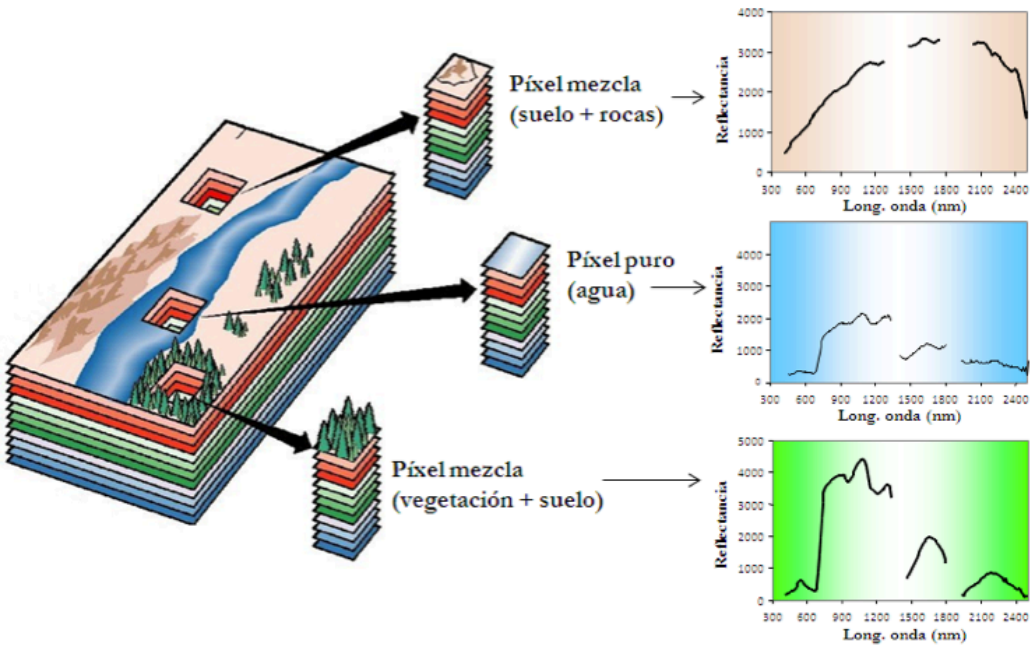


Figura 2.3: Clases de píxeles y sus firmas hiperespectrales.

La *figura 2.3* muestra un ejemplo del proceso de adquisición de píxeles puros y píxeles mezcla en imágenes hiperespectrales. También se pueden ver las distintas firmas hiperespectrales asociadas a cada tipo [6].

Aunque se ha evolucionado mucho en los instrumentos de observación remota de la Tierra, no ha sucedido en la misma media con las técnicas de tratamiento y análisis de los datos que éstos producen [7].

2.2. Sensores hiperespectrales

La obtención de imágenes hiperespectrales se produce mediante los *espectrómetros de imágenes*. Dichos instrumentos son sensores complejos cuyo desarrollo se debe a la combinación de dos tecnologías: *la espectroscopia* y *la observación remota de la superficie de la Tierra*.

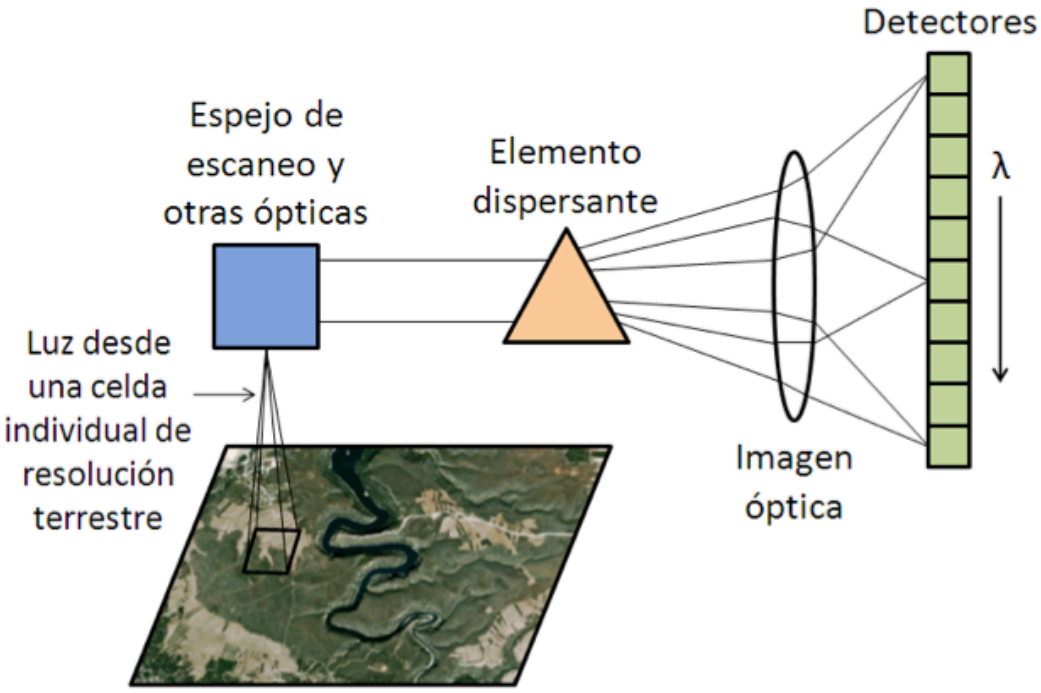


Figura 2.4: Diagrama del proceso de espectrometría de imágenes con sus elementos básicos.

La *espectroscopia* consiste en el estudio de la energía que produce la luz emitida o reflejada por parte de un material en diferentes longitudes de onda (*figura 2.4*) [8]. Las imágenes hiperespectrales, en combinación con la observación remota de la superficie terrestre, se basan en el espectro de luz solar reflejado de forma difusa

(contiene todas las longitudes de onda mezcladas) por los materiales terrestres captados en la imagen. Para realizar la dispersión de la luz en sus distintas longitudes de onda se usan los *espectrómetros*, instrumentos de medición que contienen un elemento de dispersión óptica tal como una rejilla o un prisma. Una vez separada la luz en sus distintas longitudes, la energía de cada banda es medida por detectores independientes. Mediante estos *espectrómetros* se captan las firmas espectrales de cada píxel en la imagen de un sensor hiperespectral.

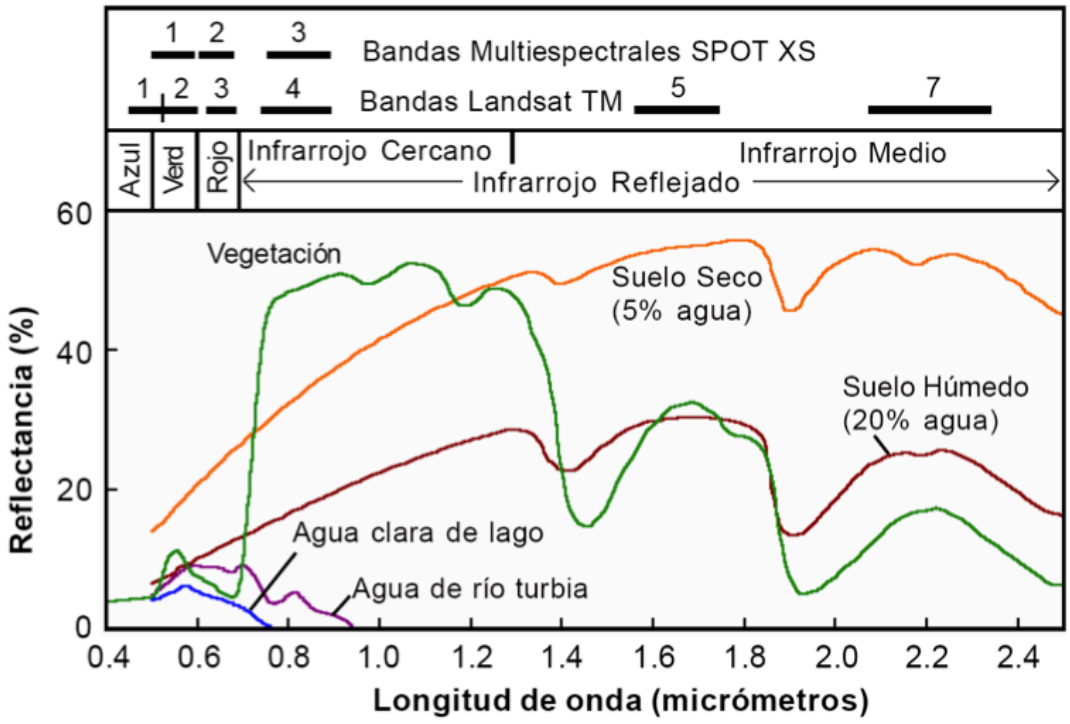


Figura 2.5: Curvas de reflectancia espectral para distintos materiales.

La propiedad fundamental en la que se basan las imágenes espectrales es la *reflectancia espectral*. Esta propiedad de la *espectrometría* consiste en observar el porcentaje de energía que se refleja sobre la energía incidente en función de la longitud de onda. La reflectancia varía según la longitud de onda y existen longitudes para las cuales la luz es absorbida o dispersada debido a los diferentes materiales que componen el área implicada. Estos puntos son conocidos como *bandas de absorción* y son característicos de cada material, por lo que, en muchos casos son utilizados para su identificación. Observando la firma espectral de un elemento en forma de gráfica, los puntos de la gráfica de reflectancia donde aparecen "valles" son las *bandas de absorción* del material en cuestión. La *figura 2.5* muestra las curvas de reflectancia espectral para diferentes materiales terrestres.

La reflectancia no es una medida directa proporcionada por el sensor. El sensor capta lo que se denomina *radiancia espectral* (cantidad de luz que capta el sensor) y para su transformación a unidades de reflectancia (porcentaje de luz reflejada sobre la emitida), hay que tener en cuenta elementos tales como la hora en que se tomó la imagen (para conocer la cantidad de luz emitida), el ángulo de incidencia de la luz, la absorción atmosférica, las condiciones meteorológicas en las que se toma la imagen, las distintas reflexiones que se producen por parte de la atmósfera, etc. [7].

2.3. Ejemplos de sensores hiperespectrales

En la actualidad ha proliferado notablemente el uso de sensores hiperespectrales para diversos propósitos. Tradicionalmente, las principales organizaciones que los usan son el *Jet Propulsion Laboratory* de la NASA y el Servicio Geológico de los Estados Unidos de América. La adquisición de las imágenes por parte de estas organizaciones se realiza mediante sensores montados en plataformas espaciales. Sin embargo, los satélites se ven muy influenciados por la meteorología de la zona y además, la toma de imágenes de un área concreta debe ser programada con antelación para poder realizar los cálculos de vuelo, obteniendo así la ruta a seguir del satélite. También deben disponer de sensores con una mayor resolución espacial, aunque las imágenes no van a ser tan detalladas como las de una aeronave debido a la distancia de la órbita.

Debido a estas limitaciones algunas organizaciones han optado por la instalación de sensores hiperespectrales en plataformas aéreas como aeroplanos, dándoles así mucha mayor flexibilidad y autonomía. Algunos ejemplos de estas organizaciones son: Norwegian Crop Research Institute (agricultura de precisión), Canada Centre for Remote Sensing (valoración de impactos ambientales), INCO Mine Limited (prospección de yacimientos minerales), etc.

A continuación, se enumeran algunos de los sensores y proyectos actuales [7]:

- ***AVIRIS*** (Airborne Visible/Infrared Imaging Spectrometer): sensor aerotransportado creado por la NASA (en uso).
- ***EO-1 Hyperion***: sensor de la NASA montado en satélite (en uso).

- ***EnMAP***: misión de captación de imágenes hiperespectrales mediante un satélite todavía en desarrollo (Alemania).
- ***PRISMA***: es una misión con un sensor hiperespectral de mediana resolución de la Agencia Espacial Italiana ASI (Agenzia Spaziale Italiana) en fase de desarrollo desde 2008.
- ***HyspIRI***: otra misión basada en el lanzamiento de un satélite. También se encuentra todavía en desarrollo (EE.UU.).

2.4. Utilidad de las imágenes hiperespectrales

Las imágenes hiperespectrales pueden aplicarse a muy diversos campos con el fin de agilizar y facilitar multitud de tareas y propósitos. Algunas de estas tareas son:

- ***Agricultura***: Aunque el coste de la adquisición de imágenes hiperespectrales es típicamente alta, para cultivos y climas específicos, el uso de sensores hiperespectrales para supervisar el desarrollo y la salud de los cultivos es cada vez mayor. Además, se está trabajando para utilizar los datos hiperespectrales con el fin de determinar la composición química de las plantas, pudiéndose utilizar estos resultados para detectar el estado de los nutrientes y las necesidades de agua en los sistemas de riego [9, 10, 11, 12].
- ***Mineralogía***: El uso de sensores hiperespectrales en este campo se encuentra mucho más implantado. Muchos minerales pueden ser identificados a partir de imágenes aéreas, y la relación de algunos de estos minerales con la presencia de minerales valiosos, tales como el oro o los diamantes, es ampliamente conocida. En la actualidad, el progreso está orientado hacia la comprensión de la relación entre el petróleo y las fugas de gas de las tuberías y pozos naturales, sus efectos sobre la vegetación y las firmas espectrales [13, 14, 15].
- ***Vigilancia***: Las imágenes hiperespectrales son particularmente útiles en la vigilancia militar debido a las contramedidas que las entidades militares ahora toman para evitar la vigilancia aérea. La idea que impulsa la vigilancia hiperespectral es que el análisis hiperespectral extrae la información a partir de una gran parte del espectro de luz en la que cualquier objeto dado debe tener una firma espectral única en al menos algunas de las muchas bandas que son escaneadas [16].

- ***Química de imágenes***: En la guerra, los ataques con agentes químicos y compuestos tóxicos industriales son algunas de las amenazas más dañinas para las tropas terrestres. De hecho, los soldados pueden estar expuestos a una amplia variedad de riesgos químicos, tanto dentro como fuera del campo de batalla. Estas amenazas son en su mayoría invisibles y muy difíciles de detectar. Sin embargo, la tecnología de imágenes hiperespectrales ofrece una capacidad única de detección e identificación para tales ataques con agentes químicos, con concentraciones de hasta sólo unas pocas partículas por millón [17].
- ***Medio ambiente***: La mayoría de los países requieren un seguimiento continuo de las emisiones producidas por el carbón, los gasoductos, los incineradores de residuos municipales, las plantas de cemento y muchos otros tipos de fuentes industriales. Este seguimiento se realiza generalmente mediante el uso de sistemas de muestreo extractivos, junto con técnicas de espectroscopia de infrarrojos. Los sensores de imágenes hiperespectrales de infrarrojos ofrecen ahora la posibilidad de obtener una imagen completa de las emisiones resultantes de chimeneas industriales desde una ubicación remota, sin necesidad del uso de sistemas de muestreo extractivos [18].

Capítulo 3

Hardware reconfigurable

Las dos formas tradicionales de realizar computación mediante hardware son: desarrollando un circuito específico para un propósito concreto con sus correspondientes interconexiones (también llamados Application Specific Integrated Circuits o ASICs) [8], o bien mediante el uso de un hardware de propósito general como un procesador y ejecutar los cálculos mediante la capa software programable de éste. Cada uno de los métodos tiene ventajas e inconvenientes. El alto rendimiento de los ASICs frente al hardware de propósito general es incuestionable, pues se trata de componentes diseñados específicamente para las operaciones a desarrollar, los cuales no tienen que ejecutar instrucciones extra como ocurre con los procesadores. Sin embargo, su principal problema es que carecen de flexibilidad para poder realizar otros cálculos. Justo aquí es donde destaca el hardware de propósito general. Otro punto importante a tener en cuenta es el coste y tiempo de desarrollo de cada uno de ellos. Crear software suele ser una tarea rápida y sencilla en comparación con la lentitud y laborioso diseño, construcción y posterior testeo de los ASICs.

El hardware reconfigurable se encuentra a medio camino de ambos e intenta ofrecer una alternativa equilibrada beneficiándose de las características positivas de ambos mundos. Por tanto con el hardware configurable se va a disponer de la eficiencia en procesamiento del hardware específico y gran parte de la flexibilidad del hardware de propósito general (*figura 3.1*) [19, 7].

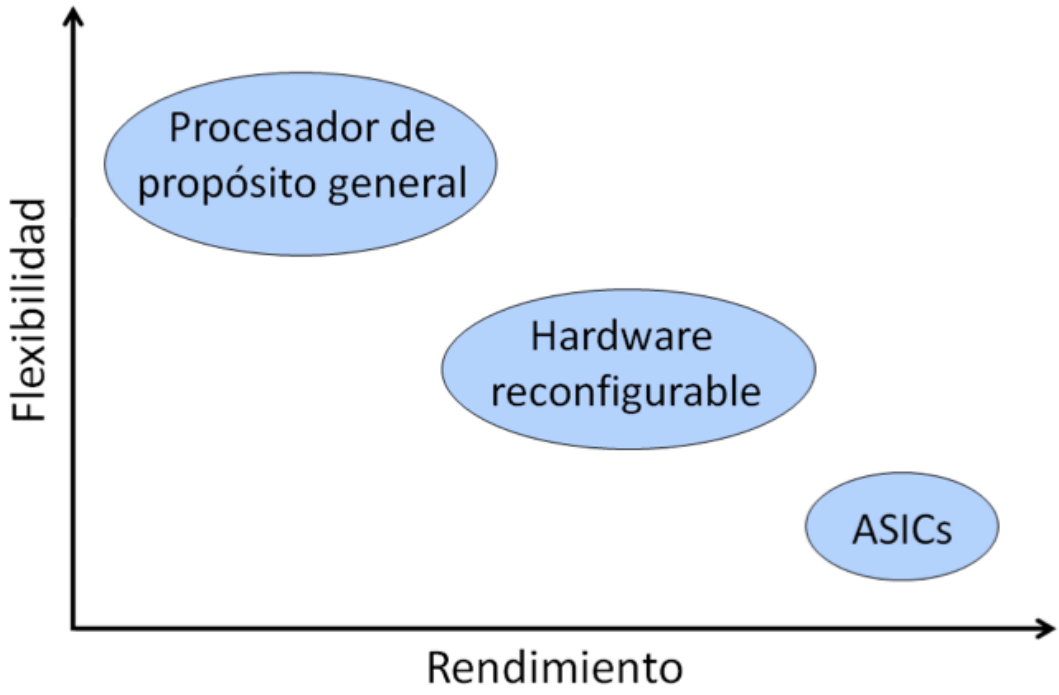


Figura 3.1: Comparación de flexibilidad y rendimiento de las distintas opciones de computación.

3.1. FPGAs como tecnología reconfigurable

A continuación se explicarán las principales características de las Field Programmable Gate Arrays (FPGAs).

La estructura básica de las FPGAs consiste en una matriz de bloques lógicos, en inglés *Logic Blocks* (LBs), y una red de interconexión de los mismos. La funcionalidad de dichos bloques y las interconexiones de la red pueden modificarse mediante un conjunto de bits de configuración que cargamos en la FPGA (*bitstream*) [20]. La configuración de la FPGA se puede realizar de dos formas: utilizando dispositivos anti-fuse [21] o mediante bits guardados en una memoria SRAM con la que se configuran los transistores de la placa como desee el programador [22]. La reprogramación del dispositivo usando anti-fuses es muy limitada, sin embargo la utilización de la SRAM es más versátil y puede permitir reconfiguración dinámica, concepto muy interesante para uno de nuestros propósitos, del cual hablaremos más adelante.

Los *Logic Blocks* (LBs) están formados a su vez por: uno o varios circuitos combinacionales programables denominados *Look-Up Tables* (LUTs), uno o más

biestables, celdas de memoria SRAM para guardar la configuración actual del LB (lo que determinará su comportamiento) y la lógica necesaria para la interconexión de todos estos componentes [23].

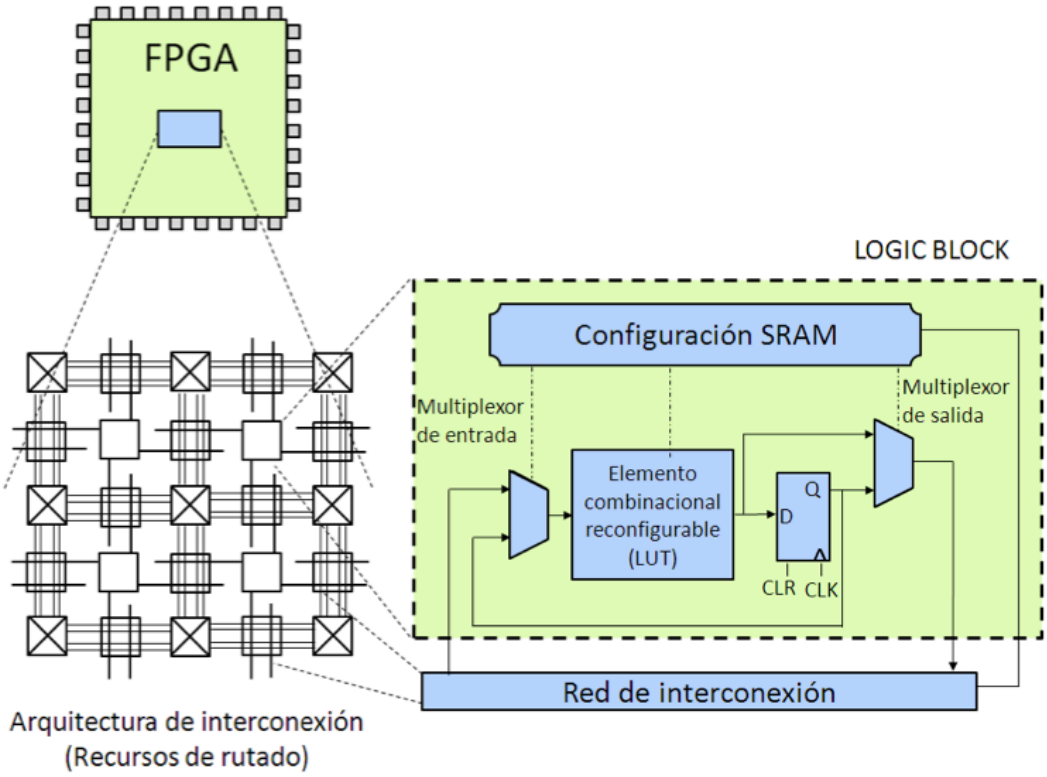


Figura 3.2: Diseño interior de una FPGA.

La comunicación con el exterior de la FPGA se realiza mediante LBs programados para dicho propósito o mediante un determinado tipo de bloques disponibles llamados *Input-Output-Blocks* (IOBs). Todos estos componentes se pueden ver de forma esquemática y modular en la *figura 3.2*.

En la actualidad existen FPGAs híbridas que integran también otros elementos como: bloques de memoria RAM, unidades aritméticas e incluso algún procesador. Una buena combinación de dichos componentes puede ser muy beneficioso en diversas aplicaciones [14].

3.2. Tipos de configuración de las FPGAs

La configuración de las FPGAs se realiza mediante el volcado en placa de una secuencia de bits de configuración a la que se denomina *bitstream*. Dichos bits determinan la funcionalidad y configuración del hardware.

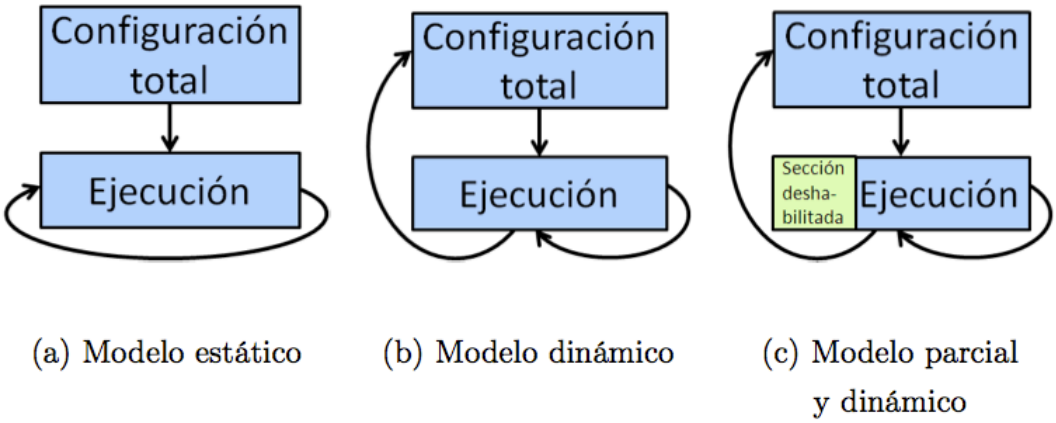


Figura 3.3: Diagramas de los distintos modos de configuración de la FPGA.

El proceso de configuración o reconfiguración es lento y costoso en comparación con la ejecución [25, 26, 27]. Los modelos de configuración más comúnmente usados son los siguientes [7]:

- ***Reconfiguración estática***: La configuración se carga en el dispositivo y una vez comenzada la ejecución no existe posibilidad de reconfigurarlo. Es la más restrictiva. Figura 3.3 (a).
- ***Reconfiguración dinámica***: se basa en el concepto de Hardware Virtual [28], de forma que se va alternando la configuración según se necesite para la ejecución de un algoritmo. Normalmente se mantiene en ejecución parte del dispositivo mientras la otra parte se reconfigura. Este tipo de reconfiguración tiene variantes según se aplique. Figura 3.3 (a) y (b):
  - ***De contexto único***: se da en FPGAs cuya memoria sólo soporta acceso secuencial. La reprogramación del dispositivo debe ser completa en cada reconfiguración. Provoca gran penalización temporal. Figura 3.4 (a).

- ***Multi-contexto***: cada componente configurable puede guardar más de una configuración y funcionar con alguna de ellas en cada momento, por tanto aumenta el número de bits necesarios en la SRAM de configuración [29, 30]. Cada una de estas configuraciones se denomina *plano*. La idea es que el cambio entre planos se realiza de forma muy rápida lo que elimina el cuello de botella del tiempo de reconfiguración. Estas FPGAs por el momento sólo son teóricas. Figura 3.4 (b).
- ***Parcial***: la más importante actualmente y la que hace de las FPGAs los dispositivos más interesantes junto con otras de sus características para los propósitos de este proyecto. En este caso se reconfigura sólo una parte del dispositivo mientras otra de las partes sigue realizando trabajo útil. El modo de uso es como el de una memoria en la que se reescribe el contenido de una región mientras otras regiones mantienen su valor. De esta forma el tiempo de reconfiguración se reduce y a la vez se sigue realizando el cómputo en las demás zonas del dispositivo [31]. Las FPGAs de la familia Virtex de Xilinx (seleccionada la Virtex-5 para este proyecto) soportan este tipo de reconfiguración dinámica [32]. Figura 3.4 (c).

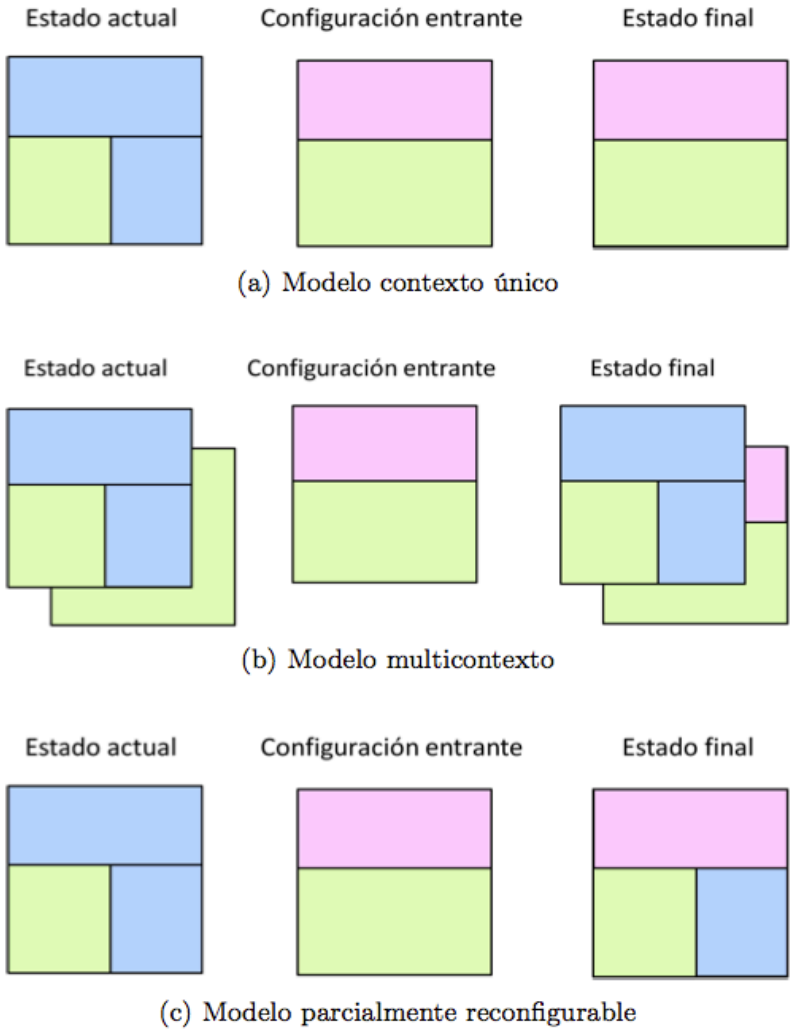


Figura 3.4: Modelos de configuración dinámica.

3.3. Diseño del hardware para FPGAs

Para el diseño del hardware, los entornos de desarrollo para FPGAs se basan en lenguajes de descripción hardware de alto nivel, en inglés *Hardware Description Language* (HDL), de los cuales los dos más importantes son Verilog [33] y VHDL [34] (*Very High-Speed Integrate Circuit Hardware Description Language*). El último que se ha mencionado, es el que se ha sido el elegido para desarrollar este proyecto.

Las ventajas de estos lenguajes son: su amplia sintaxis, la flexibilidad del lenguaje y su rápido desarrollo. Otra ventaja es la buena estructuración del lenguaje ya que de esta forma se puede hacer un diseño modular del hardware, cosa que facilita mucho la tarea del diseñador.

La selección de bloques a configurar (LBs), el interconexionado y otros matices son llevados a cabo de forma semi-automática por el sintetizador, el emplazador y el enrutador. De esta forma el programador obtiene directamente de su especificación VHDL un *bitstream* para poder volcarlo a placa y probar [7].

Otra de las ventajas importantes de usar VHDL es que se encuentra estandarizado (IEEE Std. 1076-1987) por tanto quedan reducidos problemas de compatibilidad o errores de comunicación.

3.4. Ventajas de las FPGAs

Las ventajas de usar FPGAs para computación son las siguientes [7]:

- ***Aumento de la velocidad de procesamiento***: se debe principalmente a dos factores: el *procesamiento hardware* y la *explotación del paralelismo*. En cuanto al *procesamiento hardware,* como ya se ha comentado en apartados anteriores, queda claro que un hardware específico es mucho más rápido ejecutando el algoritmo para el que ha sido diseñado que un procesador de propósito general, el cual tiene que ejecutar instrucciones extra para su funcionamiento. En cuanto a la *explotación del paralelismo,* tenemos la posibilidad de descomponer la tarea principal en subtareas que se ejecuten de forma paralela.
- ***Reducido consumo***: el bajo consumo se debe al *avance de la tecnología* empleada en la construcción de las FPGAs. Comparado con el de otras plataformas como procesadores y GPUs es bastante inferior. Xilinx sigue investigando y desarrollando nuevas tecnologías para poder seguir reduciendo el consumo [35]. Otro punto a tener en cuenta en el consumo es la *eficiencia del diseño*. Al ejecutar una tarea sólo van a estar en funcionamiento las partes necesarias en cada momento, lo que reduce al mínimo el consumo para dicho cómputo.
- ***Flexibilidad***: como ya se ha comentado, la ventaja de tener un dispositivo reconfigurable en su totalidad o en partes es muy interesante. Permite por ejemplo adaptarse en el tiempo y permitir nuevas configuraciones de aplicaciones futuras o recuperarse de un mal funcionamiento reconfigurando de nuevo el dispositivo.

- ***Coste***: gracias al gran aumento de la popularidad de las FPGAs, los precios de las mismas han bajado durante los últimos años siendo hoy en día muy competitivos con otros tipos de dispositivos.

3.5. Inconvenientes de las FPGAs

A pesar de tener grandes ventajas, como toda tecnología también tiene algunos inconvenientes. A continuación se enumeran [7]:

- ***Proceso de rutado de señales***: como ya se ha mencionado en puntos anteriores, la obtención del *bitstream* de nuestro diseño se genera a partir del código VHDL de forma automática. De esta forma, el rutado obtenido en el proceso, aunque funcional, no es el óptimo. Es por ello que es necesaria una revisión por parte del diseñador de dicho resultado. Los fabricantes de FPGAs trabajan continuamente por subsanar dicha dificultad.
- ***Tiempo de reconfiguración***: es el cuello de botella de esta tecnología. Se puede ocultar con alguna técnica que ya hemos visto (reconfiguración parcial) pero en la actualidad se encuentra en el orden de los milisegundos.

3.6. FPGAs en misiones de observación remota

La velocidad de procesamiento, el reducido consumo, la gran flexibilidad y el bajo coste hacen que el hardware reconfigurable sea una opción muy interesante y a tener en cuenta para misiones espaciales [36, 37].

También hay que destacar el ahorro que supone la capacidad de reconfiguración. Mediante ella podemos conseguir que un satélite ya en órbita cambie totalmente su configuración y se dedique a otra tarea sin necesidad de crear una nueva misión espacial para reconfigurarlos o la creación de un nuevo satélite para dar solución a la nueva tarea [27, 40, 41, 42]. Simplemente enviando al satélite el nuevo *bitstream* necesario para el cómputo y ordenar la reconfiguración de la FPGA sería suficiente.

Es muy importante también el hecho de que existan FPGAs endurecidas para radiación y certificadas para operar en el espacio [43]. Dichas FPGAs al estar certificadas incorporan corrección de errores, aspecto muy importante en una misión espacial, sobre todo si una tarea es crítica. Estas FPGAs están siendo utilizadas, en la actualidad, para diversas aplicaciones militares y espaciales.

Gracias a todas estas ventajas y a un desarrollo eficiente de los algoritmos a ejecutar, las FPGAs parecen ser la opción ideal para eliminar el cuello de botella que supone la transferencia de datos en "bruto", que toman los sensores del satélite, a la Tierra. Las FPGAs se encargan del tratamiento de los datos y obtención de resultados, de esta forma sólo se transfieren dichos resultados, con lo que tenemos un volumen de datos mucho menor que el original. En la actualidad ya existen sistemas que se dedican a la reducción y obtención de resultados a partir de estos datos, pero su consumo y coste es elevado. Las FPGAs dan solución a dicho problema [7].

En la actualidad existen ya sistemas, en uso, para realizar observación remota. Estos sistemas se basan en el uso de procesadores junto con hardware específico. Ya existe mucha flexibilidad en dichos sistemas, pero esta flexibilidad está limitada a la parte software. Las FPGAs mediante su gran flexibilidad a nivel hardware solventan dicho problema. Además, si tenemos en cuenta las nuevas FPGAs híbridas que combinan procesadores junto con un gran área reconfigurable [24, 38, 39], estaríamos contando con flexibilidad en ambos campos: software y hardware. Tampoco podemos olvidar la facilidad de desarrollo que aportan las herramientas de diseño en FPGAs en comparación al diseño de hardware específico gracias a la utilización de los lenguajes de especificación hardware (HDLs).

Todos estos aspectos hacen de las FPGAs el futuro para aplicaciones espaciales.

Capítulo 4

Detección de anomalías

La detección de anomalías es una tarea muy importante en la explotación de los datos en imágenes hiperespectrales. Los detectores de anomalías permiten encontrar en la imagen aquellas firmas espectrales que son distintas a otras en un determinado área de forma muy brusca. Estas firmas anómalas se comparan con la imagen de fondo y la probabilidad de que se produzcan es baja.

Ya existe un enfoque muy conocido para la detección de anomalías en imágenes hiperespectrales. Fue desarrollado por Reed y Yu y a esta técnica se la conoce comúnmente como *algoritmo RX* [46, 47].

El algoritmo RX está ampliamente aceptado, pero el principal problema es su alta complejidad al ser aplicado en imágenes hiperespectrales reales. Por ello se han desarrollado distintas técnicas para paliar dicha complejidad. Una de las más importantes es el uso de arquitecturas e implementaciones paralelas (clusters y procesadores multi-core) junto con estrategias de partición de datos para reducir el coste del cálculo de la matriz de covarianza, y su inversa, que es la parte más costosa del algoritmo.

4.1. Algoritmo RX

El algoritmo RX ha sido ampliamente utilizado en teoría de la señal y procesamiento de imágenes [44]. El filtro que implementa el algoritmo se conoce como filtro RX y queda definido por la siguiente expresión:

$$\delta^{RX}(x) = (x-\mu)^T K^{-1}(x-\mu)$$

donde $x = [x^{(0)}, x^{(1)}, \dots, x^{(n)}]$ es una muestra o pixel hiperespectral n-dimensional, μ es la media de la muestra de la imagen hiperespectral y K es la matriz de covarianza de los datos de la muestra. Como podemos ver, δ^{RX} es en realidad la conocida distancia de Mahalanobis [45]. Es importante tener en cuenta que las imágenes generadas por el algoritmo RX normalmente son en escala de grises, en este caso, las anomalías se pueden clasificar dependiendo del valor devuelto por el algoritmo. De esta forma el píxel con mayor valor δ^{RX}, en la imagen, puede ser considerado como la primera anomalía, y así sucesivamente. Por lo tanto, desde el punto de vista computacional, el algoritmo RX produce la siguiente salida en cuatro pasos:

1. Calcular la matriz K de covarianza de datos de la muestra $[O(\text{filas} * \text{columnas} * \text{bandas}^2)]$
2. Calcular K^{-1} por el método de Gauss $[O(\text{bandas}^3)]$
3. Calcular la distancia de Mahalanobis como δ^{RX} $[O(\text{filas} * \text{columnas} * \text{bandas}^2)]$
4. Localizar el valor máximo de δ^{RX} o anomalía $[O(\text{filas} * \text{columnas})]$

La complejidad computacional de cada paso se describe entre corchetes, donde las filas y las columnas son las dimensiones espaciales de la imagen hiperespectral de entrada y las bandas indican la dimensión espectral [46, 47].

Capítulo 5

Implementación en FPGA del algoritmo RX

En este punto se va a realizar un análisis detallado de la implementación propuesta del algoritmo RX para una plataforma FPGA. Este desarrollo supondrá ya calculada la matriz de covarianza. Dicha matriz estará almacenada en un módulo de memoria al cual se tendrá acceso para poder realizar los cálculos del algoritmo. Se propondrá primero una visión general del diseño para luego ir profundizando sobre el mismo.

5.1. Visión general de la implementación

La implementación del algoritmo cuenta con seis módulos bien diferenciados. A continuación explicamos cada uno de ellos, sus funcionalidades dentro del algoritmo y las comunicaciones entre los mismos (*Figura 5.1*).

Los módulos son los siguientes:

- ***Memory K*:** módulo de memoria utilizado para cargar la matriz de covarianza K en un inicio e ir calculando la matriz inversa mediante el método de Gauss. Una vez calculada la inversa, suponiendo que para dicho caso exista, en la memoria debe quedar la matriz identidad.
- ***Memory* $\boldsymbol{K^{-1}}$:** módulo de memoria utilizado para cargar la matriz identidad en un inicio e ir calculando la matriz inversa mediante el método de Gauss. Una vez realizado el cálculo de la inversa, suponiendo que para dicho caso exista, es en esta memoria donde se encuentra el resultado, que posteriormente se utilizará para realizar el cálculo del δ^{RX}.

- ***Inv Module*:** módulo encargado de realizar el cálculo de la inversa mediante el método de Gauss utilizando como matrices iniciales la memoria K (matriz covarianza) y la memoria K^{-1} (matriz identidad). Al finalizar el cálculo, si para los datos de entrada existe la inversa, envía una señal de finalización a la unidad de control general. En caso de no existir, envía una señal de error.
- ***Matrix mult*:** módulo encargado de realizar las multiplicaciones de matrices que aparecen en la fórmula del δ^{RX} anteriormente citada. Este multiplicador sólo puede realizar una única multiplicación de dos matrices en cada momento, por tanto cede la responsabilidad de ir cambiando las matrices a la unidad de control general.
- ***Ordered list*:** módulo encargado de guardar la lista ordenada de anomalías encontradas en la imagen a partir de los resultados obtenidos por el multiplicador de matrices (*Matrix mult*).
- ***Control unit*:** unidad de control que dirige la correcta ejecución del algoritmo mediante una máquina de estados y señales de control para los módulos. Este módulo es también el encargado de avisar de la finalización o posibles errores durante la ejecución del algoritmo.

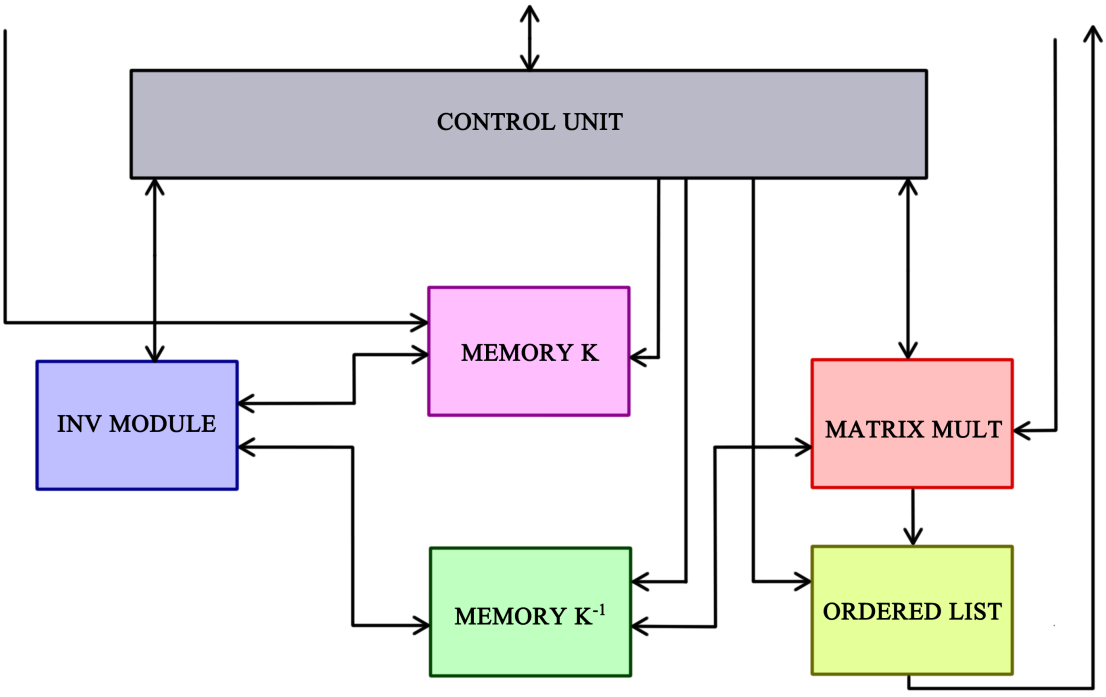


Figura 5.1: Esquema general del módulo RX.

5.2. Implementación algoritmo RX

Como ya se ha mencionado anteriormente, la implementación consta de un módulo general, denominado RX, que contiene 6 grandes submódulos en su interior.

Las entradas para este módulo general son las siguientes: una señal inicio (*start*), para comenzar con la ejecución del algoritmo; una señal *restart,* para reiniciar el algoritmo y dos señales (*dataRowK* y *dataRowXMu*) para la entrada de datos procedentes de las dos memorias externas al módulo que contienen la matriz de convarianzas y la matriz $x - \mu$ respectivamente.

Las salidas del módulo se dividen en: un señal *error* que indica cuando no se ha podido terminar el algoritmo al no tener inversa la matriz de convarianzas; una señal *finish* que se activa cuando ha terminado la ejecución del algoritmo; dos señales (*addrRowK* y *addrRowXMu*) para la petición de datos sobre las memorias externas al módulo y una última señal denominada *index_out* que contiene el resultado del algoritmo una vez se ha activado la señal *finish* mencionada anteriormente.

Los datos en la implementación se consideran como números en punto flotante con precisión simple (32 bits) siguiendo el estándar IEEE 754. Cada dato contiene un bit de signo (0 positivo, 1 negativo), un exponente de 8 bits con un sesgo de valor 127 y una mantisa de 23 bits en la que el punto binario estará a la izquierda de esos 23 bits. Realmente, la mantisa consta de 24 bits, ya que cuenta con un 1 implícito como bit más significativo aunque no ocupe una posición de bit real (*Figura 5.2*).

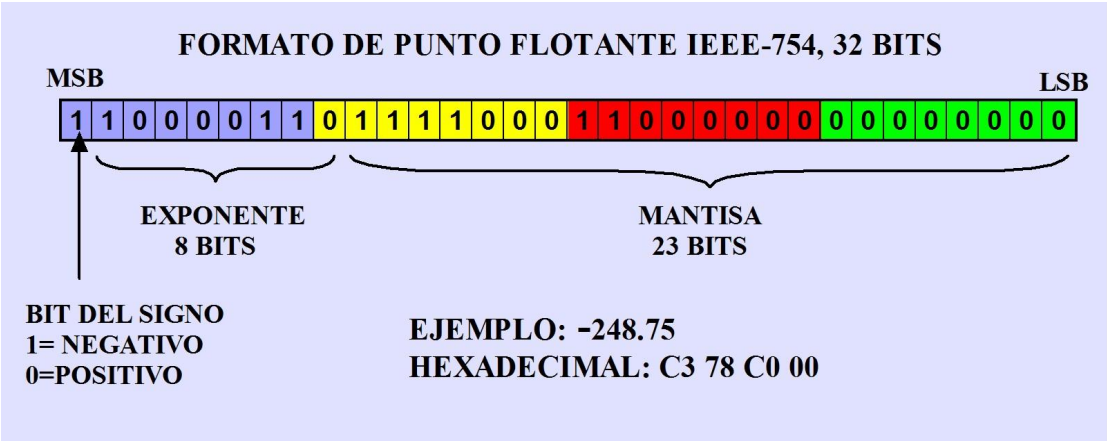


Figura 5.2: Notación en punto flotante.

A continuación se expone una imagen con el cableado de los submódulos entre sí, para en los apartados posteriores profundizar sobre la implementación de cada módulo por separado (*Figura 5.3*).

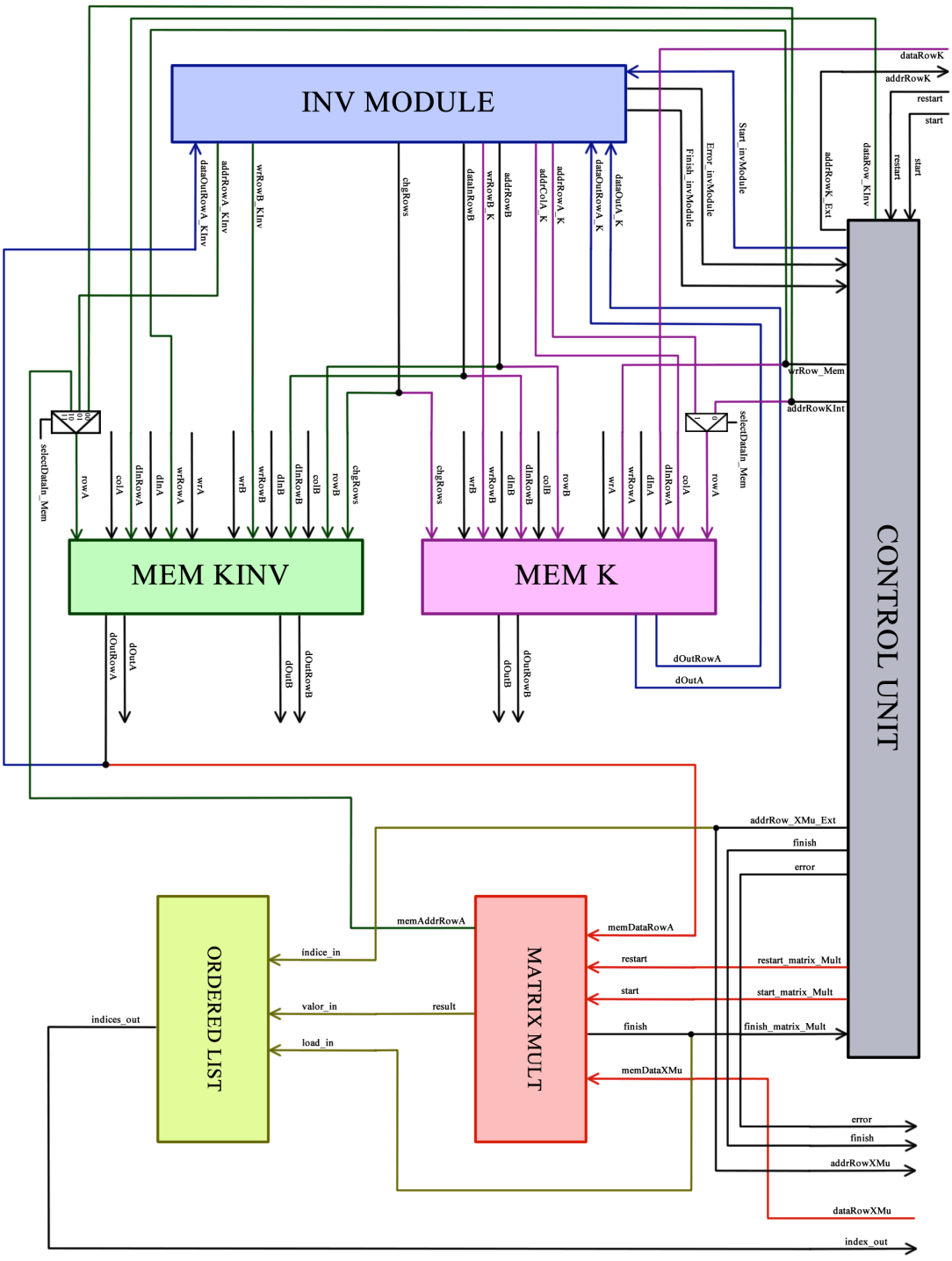


Figura 5.3: Interconexión entre los submódulos del módulo RX, sus puertos y señales.

5.2.1. Estructura de las memorias

Las memorias básicas con las que cuenta este proyecto se han diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (ipCore), creándose con las siguientes propiedades:

- Tipo de memoria: RAM de doble puerto real.
- Algoritmo para concatenar los bloques de RAM primitivos: mínimo área.
- Ancho de la memoria: 32 bits.
- Profundidad de la memoria: 512 bits.
- Modo de operar: primero escritura.

Cada memoria básica representará una columna de la memoria general. Para formar esta memoria se generan n módulos como los descritos anteriormente, obteniéndose así las n columnas de la matriz. Cabe destacar que si el resultado de multiplicar $n * 32$ es mayor que la profundidad de la memoria básica (512 bits) habría que ampliar esta misma cifra hasta que sea al menos igual a la anterior multiplicación.

La memoria básica cuenta con las siguientes entradas para cada uno de los dos puertos: *din*, dato de entrada para escribir en la memoria, *addr*, señal para direccionar la memoria y *wr* como señal de escritura.

Para poder escribir filas completas en la memoria general se utiliza como entrada un bus con n datos llamado *dInRow*. Cada dato de este bus se encuentra dirigido a una memoria básica. En el caso de que se quiera escribir un solo dato se cuenta con la señal de entrada *dIn* que se dirige a todas las memorias para luego solo escribirse en una, teniendo en cuenta la columna en la que se quiere escribir. Para escoger entre estos dos casos se tiene un selector por cada memoria básica cuya salida irá al puerto *dIn* de la misma.

La memoria general consta de dos señales de escritura: *wrRow*, para la escritura de filas completas y *wr* para la escritura de un solo dato. En el caso de que solo se quiera escribir un dato se compara si la señal *col* con la que se especifica la columna que se quiere leer o escribir corresponde con alguna de las memorias básicas. En caso afirmativo se activa la señal de escritura sólo para esa memoria básica. Si se quiere escribir un fila completa se activan todos los puertos de escritura

de las memorias. Para escribir sólo una de las señales de escritura puede estar activa, en caso contrario no se escribe durante ese ciclo (*Figura 5.4*).

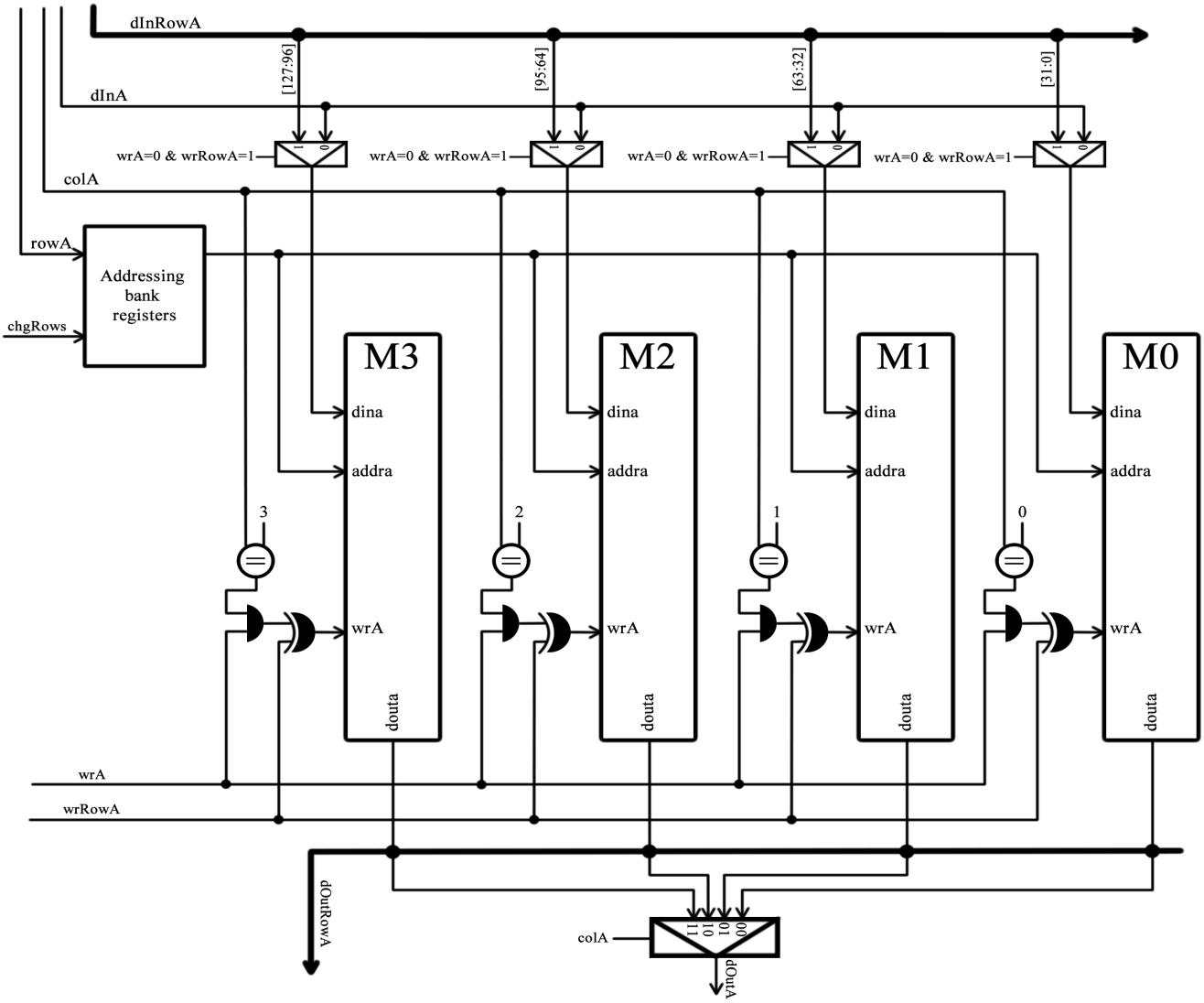


Figura 5.4: Estructura interna de las memorias utilizadas.

La salidas de datos de las memorias básicas se recogen en un bus de salida que representa una fila de la memoria general (*dOutRow*). La salida de un solo dato viene dada por un selector sobre todas las salidas de las memorias básicas cuya señal de selección se corresponde con la señal de entrada *col*.

Para realizar un cambio de filas en la memoria se dispone de un banco de registros sobre las entradas *rowA* y *rowB* de la memoria general, cuyas salidas se conectan a los puertos *dInA* y *dInB* de las memorias básicas respectivamente. El i-ésimo registro contendrá la dirección de memoria en la que se encuentra la i-ésima

fila de la matriz, por lo que para realizar un cambio de filas basta con intercambiar el contenido de estos registros entre sí.

A continuación se expone una imagen detallada sobre la estructura de una memoria general. La memoria de la imagen consta de un único puerto A, siendo el puerto B de la implementación real análogo.

Tal y como está implementado el algoritmo, la unidad de control utiliza los puertos A de las dos memorias para sus respectivas inicializaciones. El módulo *inv module* utiliza los puertos A para las lecturas de las dos matrices y los puertos B para las escrituras. El módulo *Matrix mult* usa exclusivamente el puerto A de la memoria K^{-1} para la lectura de la matriz inversa, mientras que el módulo *ordered list* no requiere de ninguna de las dos matrices para realizar sus cálculos.

5.2.2. Módulo matriz inversa

El módulo para el cálculo de la matriz inversa, necesario en el algoritmo, se divide a su vez en tres submódulos interconexionados entre sí. Estos submódulos mencionados son los siguientes (*Figura 5.5*):

- ***Data path*:** ruta de datos para realizar las operaciones entre las filas de la matriz según el método de Gauss. Precisa de la unidad de control para ejecutar los cálculos deseados.
- ***Memory Controller*:** módulo cuya tarea consiste en escribir en las memorias las filas resultantes de la ruta de datos.
- ***Control Unit*:** unidad de control encargada de la correcta sincronización entre la ruta de datos y las memorias, así como de la adecuada ejecución del método de Gauss en cada uno de sus pasos. El módulo comienza el cálculo al recibir una señal de inicio y concluye enviando una señal de finalización o de error a la unidad de control general.

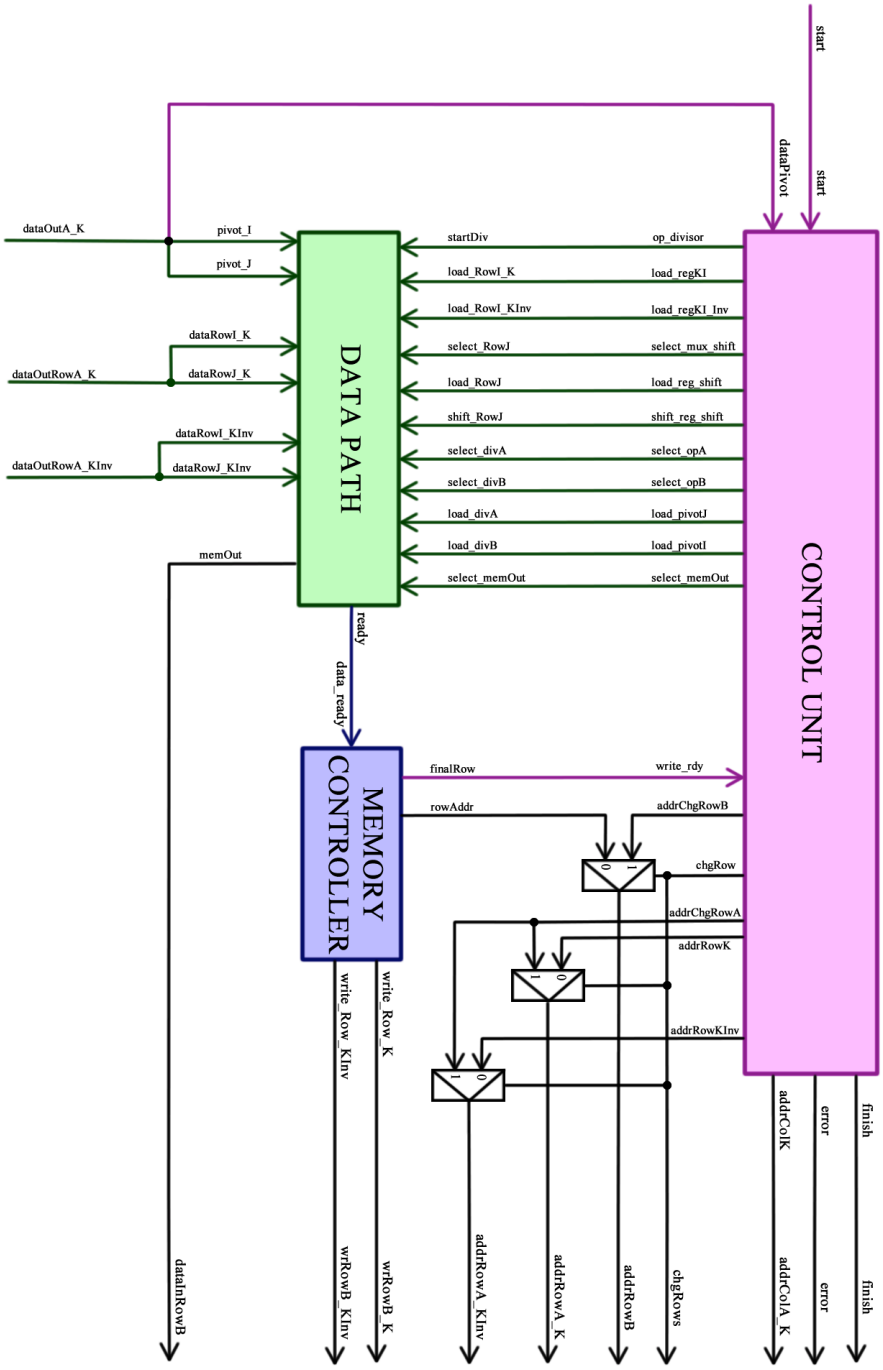


Figura 5.5: Interconexión entre los módulos que componen el módulo matriz inversa.

5.2.2.1. Método de Gauss

Se dice que una matriz cuadrada A es inversible, si existe una matriz B con la siguiente propiedad:

$$A * B = B * A = I$$

siendo I la matriz identidad. La matriz identidad es aquélla cuyos elementos son nulos salvo los de la diagonal principal, que son 1, y además dicha matriz es cuadrada. Denominamos a la matriz B la inversa de A y la denotamos por A^{-1}.

Ambos productos han de dar como resultado la matriz identidad, y ésta es cuadrada, lo que obliga a que para poder hablar de inversión de una matriz, la matriz ha de ser cuadrada. Sin embargo, es una condición necesaria pero no suficiente; no toda matriz que sea cuadrada tiene matriz inversa.

Uno de los métodos para realizar el cálculo de la matriz inversa es el método de Gauss. En términos generales, no siempre podemos garantizar que la matriz en cuestión tenga inversa, sin embargo, en caso de que sea así, utilizando este método se obtendrá la inversa sin hacer operaciones demasiado complicadas. Si la matriz no se puede invertir, el método llegará a una situación en que lo indicará.

El cálculo de la matriz inversa por el método de Gauss supone transformar una matriz en otra, equivalente por filas. La demostración rigurosa del procedimiento que a continuación se describe se sale del propósito del presente bloque, aquí se limita a su exposición.

En esencia, el método consiste (para una matriz cuadrada de orden n) en:

1. Formar una matriz de orden n x 2n tal que las primeras columnas sean las de la matriz A y las otras n las de la matriz identidad de orden n.
2. Mediante las transformaciones elementales de las filas de una matriz, convertir la matriz anterior en otra que tenga en las n primeras columnas la matriz identidad y en las n últimas otra matriz que precisamente será A^{-1}.

Por medio de las transformaciones elementales, se modifica la matriz A hasta obtener la matriz identidad. Cada paso que se aplique a la matriz se aplica también a la matriz identidad. Cuando hayamos obtenido la matriz identidad en la matriz A, la matriz B será necesariamente A^{-1}. Si no se puede llegar a la matriz identidad (por ejemplo, sale alguna fila llena de ceros), significa que la matriz no es inversible.

Las transformaciones elementales son las siguientes: sustituir una fila o columna de la matriz por ella misma multiplicada (o dividida) por un número, sustituir una fila o columna de la matriz por una combinación lineal de filas o columnas de la matriz (si trabajamos por fila, filas, y si es por columna, columnas), e intercambiar dos filas entre sí o dos columnas.

En el algoritmo implementado para la resolución de la matriz K^{-1} por el método de Gauss se utilizarán las dos últimas operaciones elementales descritas para un primer doble bucle que formará una matriz diagonal (todos los elementos situados por encima y por debajo de la diagonal principal son nulos). La primera operación elemental se utiliza en un segundo bucle que transforma esa matriz diagonal en la matriz identidad.

Para conseguir la matriz diagonal en el primer doble bucle se fija en el bucle exterior una fila pivote, a la que denominaremos fila I. El bucle interior itera *n*-1 veces cogiendo cada vez una fila distinta a la fila I, a la que denominaremos fila J. En cada iteración se formará un cero en la columna I de la fila J. Al final de la ejecución del bucle interno, la columna I resultante de la matriz K será todo ceros excepto en la fila I. El bucle exterior se ejecuta n veces para obtener al final de su ejecución la matriz diagonal.

La formación de un cero en la columna I de la fila J se consigue de la siguiente forma: se divide el elemento K[J][I] entre el elemento K[I][I] (fila pivote). Al multiplicar el resultado de la división por la fila I se obtiene una fila en cuya posición i-ésima se encuentra un elemento de igual valor que K[J][I]; luego si se realiza la resta entre la fila J y la conseguida anteriormente se obtiene un cero en la columna I de la fila J.

En el caso donde el elemento *i*-ésimo de la fila pivote (K[I][I]) sea un cero, el algoritmo comprueba si el elemento *i*-ésimo de la fila siguiente es distinto de cero. Si es distinto a cero intercambia la fila pivote que se tenía por esta nueva (realizando este mismo intercambio en la matriz K^{-1}). En el caso de que sea también cero, realiza la

comprobación con la siguiente fila. Si toda la columna restante son ceros y el algoritmo no puede intercambiar la fila pivote por otra, no se puede conseguir la matriz identidad, por lo tanto no existe la matriz inversa y se devuelve error.

Una vez obtenida la matriz diagonal, el algoritmo utiliza un bucle para transformar la esta matriz en la matriz identidad. Para ello el bucle en cada iteración toma una fila de la matriz K, a la que denominamos K[J], y la multiplica por 1/K[J][J]. Cabe recordar que en la matriz diagonal la fila j-ésima consta de ceros excepto en la posición j-ésima. Realizando esta multiplicación se consigue obtener una fila de la matriz identidad e iterando el bucle n veces obtenemos dicha matriz.

Como ya se había mencionado anteriormente, cualquier operación realizada sobre K ha de hacerse análogamente sobre la matriz K^{-1}, por lo que se obtendrá en la finalización de la ejecución del segundo bucle la matriz inversa de K donde antes se encontraba la matriz identidad inicial.

Para la correcta comprensión del algoritmo expuesto se muestra a continuación un ejemplo del mismo:

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \mathbf{1} & \mathbf{2} & \mathbf{-1} & \mathbf{2} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{2} & \mathbf{2} & \mathbf{-1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{-1} & \mathbf{-1} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{2} & \mathbf{1} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ (\boldsymbol{i\ actual}) \\ \boldsymbol{F2-2F1} \\ \boldsymbol{F3-(-1)F1} \\ \boldsymbol{F4-2F1} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \mathbf{1} & \mathbf{2} & \mathbf{-1} & \mathbf{2} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{-2} & \mathbf{1} & \mathbf{-3} & \mathbf{-2} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{-3} & \mathbf{1} & \mathbf{-2} & \mathbf{-2} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \boldsymbol{F1-(-1)F2} \\ (\boldsymbol{i\ actual}) \\ \boldsymbol{F3-(-0.5)F2} \\ \boldsymbol{F4-1.5F2} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{-1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{-2} & \mathbf{1} & \mathbf{-3} & \mathbf{-2} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0.5} & \mathbf{-0.5} & \mathbf{0} & \mathbf{0.5} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{-0.5} & \mathbf{2.5} & \mathbf{1} & \mathbf{-1.5} & \mathbf{0} & \mathbf{1} \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \boldsymbol{F1-0F3} \\ \boldsymbol{F2-2F3} \\ (\boldsymbol{i\ actual}) \\ \boldsymbol{F4-(-1)F3} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \hline 1 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & -2 & 0 & -2 & -2 & 0 & -2 & 0 \\ 0 & 0 & 0.5 & -0.5 & 0 & 0.5 & 1 & 0 \\ 0 & 0 & 0 & 2 & 1 & -1 & 1 & 1 \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \boldsymbol{F1-(-0.5)F4} \\ \boldsymbol{F2-(-1)F4} \\ \boldsymbol{F3-(-0.25)F4} \\ \boldsymbol{(i\ actual)} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \hline 1 & 0 & 0 & 0 & -0.5 & 0.5 & 0.5 & 0.5 \\ 0 & -2 & 0 & 0 & -1 & -1 & -1 & 1 \\ 0 & 0 & 0.5 & 0 & 0.25 & 0.25 & 1.25 & 0.25 \\ 0 & 0 & 0 & 2 & 1 & -1 & 1 & 1 \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \\ \boldsymbol{Matriz\ resultado} \\ \boldsymbol{primer\ bucle\ FOR} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \hline 1 & 0 & 0 & 0 & -0.5 & 0.5 & 0.5 & 0.5 \\ 0 & -2 & 0 & 0 & -1 & -1 & -1 & 1 \\ 0 & 0 & 0.5 & 0 & 0.25 & 0.25 & 1.25 & 0.25 \\ 0 & 0 & 0 & 2 & 1 & -1 & 1 & 1 \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \boldsymbol{F1*1} \\ \boldsymbol{F2/(-2)} \\ \boldsymbol{F3*2} \\ \boldsymbol{F4/2} \end{array}$$

$$\left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\boldsymbol{K}} & \multicolumn{4}{c}{\boldsymbol{K^{-1}}} \\ \hline 1 & 0 & 0 & 0 & -0.5 & 0.5 & 0.5 & 0.5 \\ 0 & 1 & 0 & 0 & 0.5 & 0.5 & 0.5 & -0.5 \\ 0 & 0 & 1 & 0 & 0.5 & 0.5 & 2.5 & 0.5 \\ 0 & 0 & 0 & 1 & 0.5 & -0.5 & 0.5 & 0.5 \end{array}\right) \quad \begin{array}{c} \boldsymbol{Operaciones} \\ \\ \boldsymbol{Matriz\ resultado} \end{array}$$

5.2.2.2. Ruta de datos

La ruta de datos está compuesta por los módulos básicos que se muestran en la imagen. Se va a explicar la utilidad de cada uno de dichos módulos.

Para realizar las operaciones de división del método de Gauss se dispone de un divisor de 32 bits de punto flotante de simple precisión (PF). Este divisor está diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (*ipCore*). Se ha creado con las siguientes propiedades:

- *Latencia*: 5 ciclos
- *Ciclos por operación*: 1 ciclo
- *Puertos de entrada*: 2 (A y B)
- *Puertos de salida*: 1 (resultado)
- *Puertos de control*:

  - *operation_nd*: puerto de entrada que indica al divisor cuándo debe operar con los valores que tiene en sus entradas.
  - *ready*: puerto de salida que indica la conclusión de la operación.
- *Operación a realizar*: A entre B
- *Precisión de punto flotante*: simple

Por la entrada del numerador se tiene un selector para los diferentes casos que se nos plantean:

1. ***Primer bucle***: entran los datos del elemento situado en la fila actual que estamos tratando y misma columna que el pivote. Disponemos de dos entradas en el selector para este caso: entrada directa (primer uso) o de registro (guarda el dato para posteriores usos).
2. ***Segundo bucle***: se introduce un 1 (en punto flotante) para obtener el valor que multiplicado por el pivote da un uno y así obtener la diagonal de la matriz identidad.

Por la entrada del denominador tenemos un selector que elige entre coger el pivote directo (primer uso) o de un registro (guarda el pivote para posteriores usos).

Para realizar las operaciones de multiplicación del método de Gauss se dispone de una red iterativa de multiplicadores de 32 bits cada uno donde el número de multiplicadores es igual al orden de la matriz. Esta red paraleliza las multiplicaciones de toda una fila, consiguiendo así mejorar el tiempo de ejecución frente a una versión serie. Cada multiplicador está diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (*ipCore*). Se han creado con las siguientes propiedades:

- *Latencia*: 1 ciclos
- *Ciclos por operación*: 1 ciclo
- *Puertos de entrada*: 2 (A y B)
- *Puertos de salida*: 1 (resultado)
- *Puertos de control*:
  - *operation_nd*: puerto de entrada que indica al multiplicador cuándo debe operar con los valores que tiene en sus entradas.
  - *ready*: puerto de salida que indica la conclusión de la operación.

- *Operación a realizar*: A * B
- *Precisión de punto flotante*: simple
- *Optimización*: máximo uso (3*DSP48E1)

Por los puertos B de los multiplicadores entra el valor obtenido en el divisor.

Por cada puerto A tenemos un selector con los diferentes casos a distinguir:

1. ***Primer bucle***: en este caso, tenemos otro selector con que se alterna según un contador de 1 bit cíclico disparado por el *ready* del divisor. Con este selector decidimos si los datos que llegan son la fila pivote de la memoria K o de la memoria K^{-1}. Dichas filas vienen de un registro que los almacena para su uso durante la ejecución.
2. ***Segundo bucle***: para esta situación, cogemos el dato de una de las salidas del banco de registros de desplazamiento (el cual se explica más adelante), que contendrá en un primer ciclo la fila de la memoria K y en el siguiente ciclo la fila de la memoria K^{-1}. Esta operación producirá una fila de la matriz identidad para la memoria K y una fila de resultado final para la memoria K^{-1}.

La salida de los multiplicadores está conectada a dos sitios:

1. A la entrada B de los restadores (explicados a continuación).
2. A una de las entradas de un selector que decide el valor de salida de memOut (resultado que va a memoria). Dicho selector saca el valor de los multiplicadores sólo si nos encontramos en el segundo bucle.

Para realizar las operaciones de resta del método de Gauss se dispone de una fila de restadores de 32 bits cada uno. El número de restadores depende del tamaño de la matriz. Esta disposición paraleliza las restas de toda una fila. Consiguiendo ahorrar tiempo en el cálculo. Cada restador está diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (*ipCore*). Se han creado con las siguientes propiedades:

- *Latencia*: 1 ciclos
- *Ciclos por operación*: 1 ciclo
- *Puertos de entrada*: 2 (A y B)

- *Puertos de salida*: 1 (resultado)
- *Puertos de control*:
  - *operation_nd*: puerto de entrada que indica al restador cuándo debe operar con los valores que tiene en sus entradas.
  - *ready*: puerto de salida que indica la conclusión de la operación.
- *Operación a realizar*: A - B
- *Precisión de punto flotante*: simple
- *Optimización*: máximo uso (2* DSP48E1)

Estos restadores sólo se utilizan en el primer bucle del método de la inversa por Gauss. Su salida son los resultados parciales del mismo y está conectada al selector de memOut para su escritura en memoria.

Los datos de entrada del puerto B de los restadores son los resultados de los multiplicadores. Los datos de entrada del puerto A de los restadores corresponden con una de las salidas del banco de registros de desplazamiento, que contendrá en un primer ciclo la fila de la memoria K y en el siguiente ciclo la fila de la memoria K^{-1}. Esta operación producirá una fila con un cero en la columna del pivote y una fila de resultado parcial para la memoria K^{-1}.

La señal *opera* de los multiplicadores viene de la señal *ready* que genera el divisor. A su vez, la señal *opera* de los restadores viene de las señales de *ready* generadas por los multiplicadores. Por lo tanto tenemos una estructura en *cascada*.

El banco de registros de desplazamiento consta de seis registros en batería. La salida del registro predecesor (i-1) es la entrada del registro actual (i). El desplazamiento de dichos registros se realiza en cada ciclo. La entrada del banco se alterna mediante un selector y se carga de forma alternada las filas de la matriz K y de la matriz K^{-1}cuando proceda. Este banco realiza la función de retardar los datos para la correcta sincronización. De esta forma, si sumamos la latencia del divisor y los multiplicadores en *cascada,* anteriormente mencionados, obtenemos que dicho retardo es seis (justo el número de registros de los que consta el banco). La salida del sexto registro sirve como entrada para el puerto A de los restadores (primer bucle) y la salida del quinto registro sirve como entrada, para uno de los casos, del puerto A de los multiplicadores (segundo bucle).

Todas las interconexiones, puertos y señales mencionados se ven reflejados en la *Figura 5.6*.

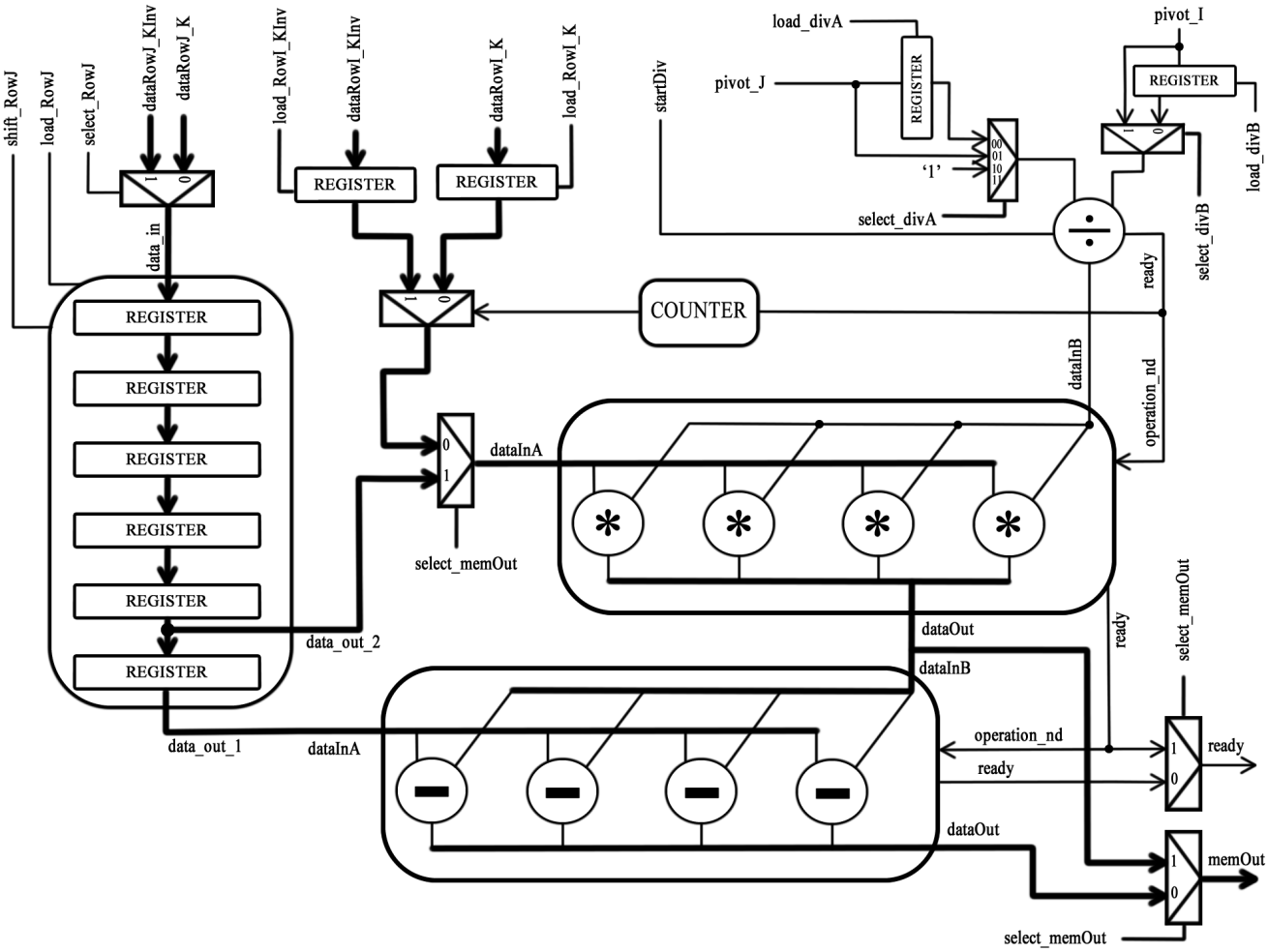
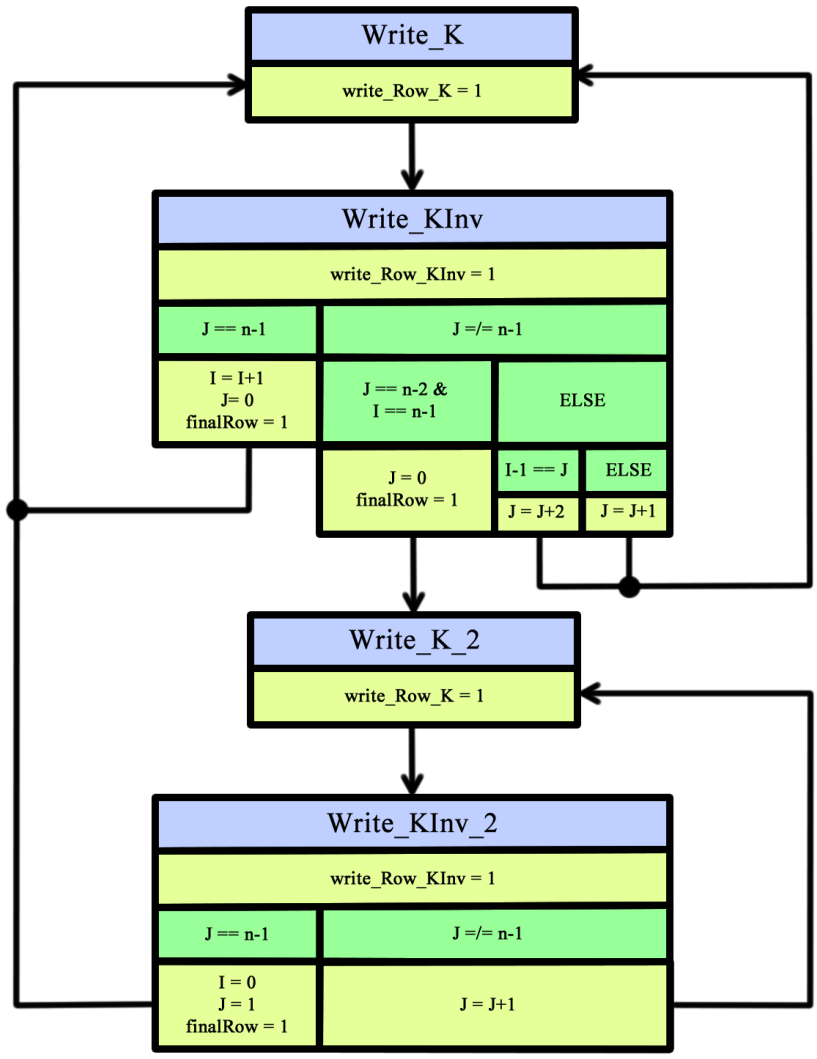


Figura 5.6: Ejemplo de ruta de datos correspondiente al módulo matriz inversa para un tamaño de matriz 4x4.

5.2.2.3. Controlador de memoria

Este submódulo se encarga de gestionar las escrituras en memoria de las filas resultado obtenidas en la ruta de datos del módulo para el cálculo de la matriz inversa. La máquina de estados asociada a este controlador sólo cambia si se produce una señal *ready* en la ruta de datos. Se diferencian claramente los dos bucles correspondientes del algoritmo implementado para el método de Gauss.

Otra funcionalidad del controlador de memoria es la de avisar a la unidad de control del cálculo de la inversa de la finalización de la iteración actual del bucle externo para así poder comenzar con la siguiente.



Leyenda

Azul *: nombre de los estados*
Amarillo: *operaciones a realizar (dependen de las condiciones que tengan por encima)*
Verde *: sentencias condicionales (estas sentencias afectan a las operaciones que se encuentren debajo de ellas)*

Figura 5.7: Diagrama de estados del controlador de memoria.

Posee una única señal de entrada *data_rdy* que se activa con la finalización de uno de los cálculos de la ruta de datos. Se comunica con las memorias mediante las señales *rowAddr* (señal de dirección de las memorias), que denominaremos a partir de

este momento como J, *write_Row_K* y *write_Row_KInv* (señales de escritura para cada memoria). Por último contiene otra señal para avisar a la unidad de control del módulo inversa de la finalización de una iteración del bucle interno o del segundo bucle.

El reseteo de la máquina de estados asociada a este controlador produce la inicialización de la variable I a cero, de la variable J a uno y asigna como estado inicial *Write_K*. Posee un total de 4 estados (*Figura 5.7*). A continuación se explica detalladamente cada uno de ellos, teniendo en cuenta, como ya se ha mencionado anteriormente, que solo se activa este módulo si se produce una señal *ready* en la ruta de datos:

- ***Write_K*:** Este estado se encarga de realizar las escrituras de la memoria K en el primer doble bucle del algoritmo de la inversa. Activa la señal *write_Row_K* y genera una transición hacia el estado *Write_KInv* para la escritura de la misma fila en la K inversa pero con los datos generados para dicha matriz.
- ***Write_KInv*:** Realiza las escrituras de la fila inversa asociada la fila escrita en el anterior estado activando la señal *write_Row_KInv*. En este punto se contemplan los siguientes casos:
  - Si J es igual a n-1 entonces se ha terminado una iteración del bucle externo por lo que se resetea la variable J y se incrementa en una unidad la variable I. Así mismo se manda la señal de finalización de escrituras *finalRow* para la unidad de control del módulo inversa. Por último volvemos al estado anterior a la espera de más escrituras de la siguiente iteración del bucle externo.
  - Si J es distinto de n-1 distinguimos otros dos casos:
    - Si J es igual a n-2 e I es igual a n-1 entonces habremos terminado de iterar el bucle externo. Reseteamos la variable J para su posterior uso en el segundo bucle, activamos la señal *finalRow* como consecuencia de haber terminado una iteración del bucle externo y realizamos la transición al estado *Write_K_2*. Que se corresponde con uno de los estados del segundo bucle del algoritmo.
    - Cualquier otro caso significa que no se ha terminado de iterar el bucle interno por lo que se aumenta en una unidad la variable J. Un suceso excepcional sería si J resulta ser igual a I-1, en cuyo

caso en vez de aumentar J en una unidad se aumentaría en dos, para así poder saltar la fila pivote. Indistintamente se realizará la transición al estado *Write_K* para seguir iterando el bucle interno.

- ***Write_K_2*:** La funcionalidad de este estado es la de realizar las escrituras correspondientes sobre la matriz K en el segundo bucle del algoritmo. Para ello activa la señal *write_Row_K* y genera una transición hacia el estado *Write_KInv_2* para la escritura de la misma fila en la K inversa pero con los datos generados para dicha matriz.
- ***Write_KInv_2*:** Realiza las escrituras de la fila inversa asociada, la fila escrita en el anterior estado activando la señal *write_Row_KInv*. Posteriormente se distinguen dos casos:
  - Si J es igual a n-1 entonces se ha llegado al final del segundo bucle. Se reinician los valores de las variables I y J, se manda la señal de finalización de escrituras *finalRow* para la unidad de control del módulo inversa y se vuelve al estado *Write_K* para comenzar de nuevo.
  - Si J es distinto a n-1 habrá que seguir iterando el segundo bucle, por lo que se aumenta en una unidad la variable J y se produce una transición al estado *Write_K_2*.

5.2.2.4. Unidad de control

La unidad de control para el cálculo de la inversa es la más compleja del proyecto. Está compuesta por un total de diecisiete estados. Es la encargada de gestionar correctamente la ruta de datos para el cálculo de la inversa.

Esta unidad empieza a realizar los cálculos con la llegada de una señal de comienzo, la cual proviene de la unidad de control principal del RX. La unidad de control también necesita el elemento i-ésimo de la fila pivote para poder realizar su comparación con cero.

El controlador de memoria envía una señal a esta unidad para avisar cuando se ha llegado ha completado una de las iteraciones del bucle externo para permitir la actualización de los índices. Este caso ya ha sido mencionado en el apartado del controlador de memoria (5.2.2.3).

Este submódulo avisa a la unidad de control principal de su terminación o si se ha producido un error en su ejecución (no se ha conseguido obtener la matriz inversa).

Se comunica con las memorias para solicitar las filas que son utilizadas en la ruta de datos. Tiene una señal que indica a las memorias cuándo deben realizar un cambio de filas en su matriz, indicando las filas implicadas.

Por último, posee numerosas señales para el control de la ruta de datos.

En la máquina de estados el acceso a los datos de las memorias tiene un retardo de un ciclo que se ha de tener en cuenta para mandar las direcciones que se desean en el momento adecuado. Dado que se trabaja con elementos secuenciales, la actualización de las direcciones que se envían a las memorias se retarda otro ciclo más. Como consecuencia de lo anteriormente citado, cuando se desee leer un dato en memoria se realizarán los siguientes pasos:

1. ***Preparar lectura [prepara]***: se actualiza la dirección que se desea leer (hasta el siguiente ciclo no le llega a la memoria).
2. ***Realizando lectura [leyendo]***: llega la dirección preparada en ciclo anterior y se utiliza el ciclo actual para obtener los datos de la memoria (hasta el siguiente ciclo no estarán disponibles).
3. ***Se tiene el dato disponible [disponible]***: La memoria ya ha puesto los datos en su salida y pueden ser utilizados.

La unidad de control cuando se resetea fija el valor de I a 0 y el de J a 1 para dejar preparada su configuración. Además, el valor de la I se manda por defecto a las memorias para su lectura, por lo que cuando llega el momento de ejecutar el algoritmo de esta unidad sólo habrá que esperar un ciclo de retardo (*leyendo*).

A continuación se explica detalladamente cada uno de los estados de la unidad de control:

- ***Initial_State:*** (estado inicial) se encarga de realizar la espera hasta la llegada de la señal de comienzo para iniciar el cálculo.
- ***Read_Row_Izero:*** (lectura de la fila I[0]) *leyendo(*$K[I], K^{-1}[I], K[I][I]$*) prepara(*$K[J], K[J][I]$*)* estado utilizado para la espera de la lectura de la primera fila I. Preparación de la lectura de la primera fila J.

- ***Read_Row_J:*** (lectura de fila J) *disponible(* $K[I], K^{-1}[I], K[I][I]$ *), leyendo(*$K[J], K[J][I]$*)* Este estado tiene distintos casos para las siguientes condiciones:

  El elemento i-ésimo de la fila I es distinto de cero. No nos encontramos en la situación de realizar un cambio de filas por lo que seguimos con la ejecución normal del algoritmo cargando la fila I de la memoria K y K^{-1} en los registros y realizando la acción *prepara(*$K^{-1}[J]$*)*. Acto seguido se transita al estado *Operate_K*, con el que se comenzarán las operaciones sobre la fila.

  - El elemento i-ésimo de la fila I es igual a cero. Nos encontramos en la situación de realizar un cambio de filas. Para ello distinguimos dos casos:
    - La fila I es la última fila de la matriz (n-1). No se puede intercambiar esta fila por ninguna otra de la matriz sin que su i-ésimo elemento sea cero, por lo que al no poderse hallar la matriz inversa el siguiente estado será el estado de error *Error_State*.
    - La fila I no resulta ser la última fila de la matriz. Se intenta buscar una fila con la que intercambiar la fila I para que su i-ésimo elemento no valga cero. Para ello se asigna a una variable llamada IAux el valor de I+1 y se realiza la acción *preparar(*$K[IAux], K^{-1}[IAux], K[IAux][I]$*)*. Por último se realiza la transición al estado *Read_Row_Chg*.

- **Operate_K:** (opera K) *disponible(*$K[J], K[J][I]$*) leyendo(*$K^{-1}[J]$*)* En este estado se manda operar al divisor para obtener un cero en la columna I de la fila J. Casos que se dan:

  - Si se ha terminado una iteración del bucle externo (no siendo la última) incrementamos la I y reseteamos J a 0. El siguiente estado siempre será *Next_Row_I*.
  - En el caso de que se hayan concluido todas las operaciones del bucle externo pasamos a un estado (*Last_Op* y *Wait_Write_2*) en que se espera a las escrituras de ésta última iteración para dar paso posteriormente al inicio del segundo y último bucle.
  - En cualquier otro caso (ninguno de los anteriores) se deberá de seguir iterando en el bucle interno distinguiendo dos subcasos:

- La siguiente fila a tratar coincide con la fila pivote, por tanto debemos saltarla en el cálculo. Debemos *preparar(* $K[J+2], K[J+2][I]$*)*.
- Si no es el caso anterior simplemente incrementamos la J, siguiendo así con la ejecución normal del bucle interno. Debemos *preparar(*$K[J+1], K[J+1][I]$*)*.

Cualquiera de estos dos subcasos llevan al estado *Next_Row_J.*

- **Next_Row_J:** diponible($K^{-1}[J]$), leyendo($K[J], K[J][I]$ teniendo J el valor dado por los subcasos anteriores), prepara($K^{-1}[I]$). En este estado mandamos operar al divisor con los mismos datos que en el estado anterior pero en caso para hacer cálculo sobre la fila de K^{-1}. Tras este estado se vuelve a Operate_K para operar la siguiente fila J (concluida una iteración del bucle interno).
- **Next_Row_I:** disponible($K^{-1}[J]$), prepara($K[I], K^{-1}[I], K[I][I]$). En este estado se manda operar al divisor con los mismos datos que en el estado Operate_K pero en caso para hacer cálculo sobre la fila de K^{-1}. Tras este estado se pasa a un estado que espera a la finalización de las escrituras de las filas calculadas (Wait_Write_1). Tras este estado queda concluida una iteración del bucle externo.
- **Last_Op:** disponible($K^{-1}[J]$), prepara($K[0], K[0][0]$). En este estado se manda operar al divisor con los mismos datos que en el estado Operate_K pero en caso para hacer cálculo sobre la fila de K^{-1}. Tras este estado se pasa a un estado que espera a la finalización de las escrituras de las filas calculadas para la última iteración del bucle externo (Wait_Write_2). Tras este estado quedan concluidas todas las iteraciones del bucle externo.
- **Wait_Write_1:** leyendo($K[I], K^{-1}[I], K[I][I]$). Es un estado que espera a la finalización de las escrituras por parte del controlador de memoria descrito en el punto anterior. Mientras el controlador de memoria no envíe la señal de finalización de las escrituras (write_rdy igual a uno) se continuará en el mismo estado, siempre preparando la lectura de la de la siguiente fila I de la matriz para su posterior uso (preparar($K[I], K^{-1}[I], K[I][I]$)). Si la señal de write_rdy se activa, se distinguen dos casos:

- La siguiente fila I a tratar no es la última fila de la matriz. Se realiza la acción *preparar(*$K[J], K^{-1}[I], K[J][I]$*)* y se transita al estado *Read_Row_J* descrito anteriormente.
- La siguiente fila I a tratar es la última fila de la matriz. Al haber terminado de escribir la última fila el controlador de memoria que ahora va a necesitar la ruta de datos como fila pivote, la fila que se estaba leyendo estaba desactualizada por lo que se necesita un estado de transición para el retardo de la lectura de memoria. Se realiza la acción p*reparar(*$K[I], K^{-1}[I], K[I][I]$*)* de la nueva fila actualizada y se transita al estado *Read_Row_Izero* que, aunque su nombre no describe el comportamiento actual de la ejecución, su contenido si se adecúa para el correcto funcionamiento.

- **Read_Row_Chg:** leyendo($K[IAux], K^{-1}[IAux], K[IAux][I]$), prepara($K[IAux+1], K^{-1}[IAux+1], K[IAux+1][I]$). Estado de transición para obtener los datos necesarios para el cambio de filas. Se utiliza un índice auxiliar (IAux) que indica la fila que se compara con cero para su posterior cambio con la fila I. Tras este estado se realiza la transición a Verificate_ChgRow.
- ***Verificate_ChgRow:*** *disponible(* $K[IAux], K^{-1}[IAux], K[IAux][I]$ *), leyendo(*$K[IAux+1], K^{-1}[IAux+1], K[IAux+1][I]$*)*. Es el estado que verifica si el elemento I de la fila IAux es distinto de cero. Casos posibles:
  - El elemento I de la fila IAux es distinto de cero. Se realiza *prepara(*$K[J], K[J][I]$*)*. Se carga la nueva fila I en los registros de la ruta de datos (tanto para K como para K^{-1}) y se sustituye en la memoria la nueva fila I (K[IAux] y K^{-1}[IAux]) por la antigua fila I (K[I] y K^{-1}[I]). Tras este estado se realiza la transición al estado *End_ChgRow*.
  - El elemento I de la fila IAux es igual a cero. Aquí se contemplan los dos siguientes casos:
    - Si IAux es menor que n-1 significa que no hemos llegado al final de las posibilidades de cambio, por tanto, se incrementa IAux para continuar con la búsqueda de una fila cuyo elemento i-ésimo no contenga un cero.
    - Si IAux es igual a n-1 implica que hemos llegado a la última posibilidad por consiguiente no podemos encontrar una fila cuyo

elemento i-ésimo sea distinto de cero.De todo ello se deduce que no existe matriz inversa para la matriz K. Se realiza la transición al estado de error *Error_State*.

- ***End_ChgRow:*** *leyendo($K[J], K[J][I]$), prepara($K^{-1}[J]$).* Estado de transición necesario para el correcto funcionamiento de la máquina de estados, se produce por el retardo en las lecturas de datos de las memorias. Se ejecuta la lectura de la siguiente fila J a operar y se realiza la petición de lectura de la misma fila pero en este caso de la matriz K^{-1}. Se realiza la transición al estado *Operate_K para* empezar con la ejecución del bucle interno.
- ***Wait_Write_2:*** *leyendo($K[0], K[0][0]$)* Es un estado que espera a la finalización de las escrituras por parte del controlador de memoria descrito en el punto anterior. La distinción entre *Wait_Write_1* y este mismo estado se produce por la necesidad de distinguir las últimas escrituras del doble bucle, para posteriormente realizar la transición a estados que realicen los cálculos del segundo bucle. Mientras el controlador de memoria no envíe la señal de finalización de las escrituras (*write_rdy* igual a uno) se continuará en el mismo estado, siempre preparando la lectura de la primera fila de la matriz para su posterior uso en el siguiente estado (p*repara($K[0], K[0][0]$)*). Si la señal de *write_rdy* se activa, se realiza la transición al estado *Divide_Row_K*.
- ***Divide_Row_K:*** *disponible($K[J], K[J][J]$), leyendo($K^{-1}[J]$), prepara($K[J+1], K[J+1][J+1]$).* La funcionalidad del este estado es la de mandar realizar los cálculos pertinentes para transformar la matriz diagonal resultante K después del primer bucle en la matriz identidad. Para ello carga la fila a tratar en los registros de desplazamiento y manda operar al divisor con el elemento J de la fila en el denominador y un 1 en el numerador.

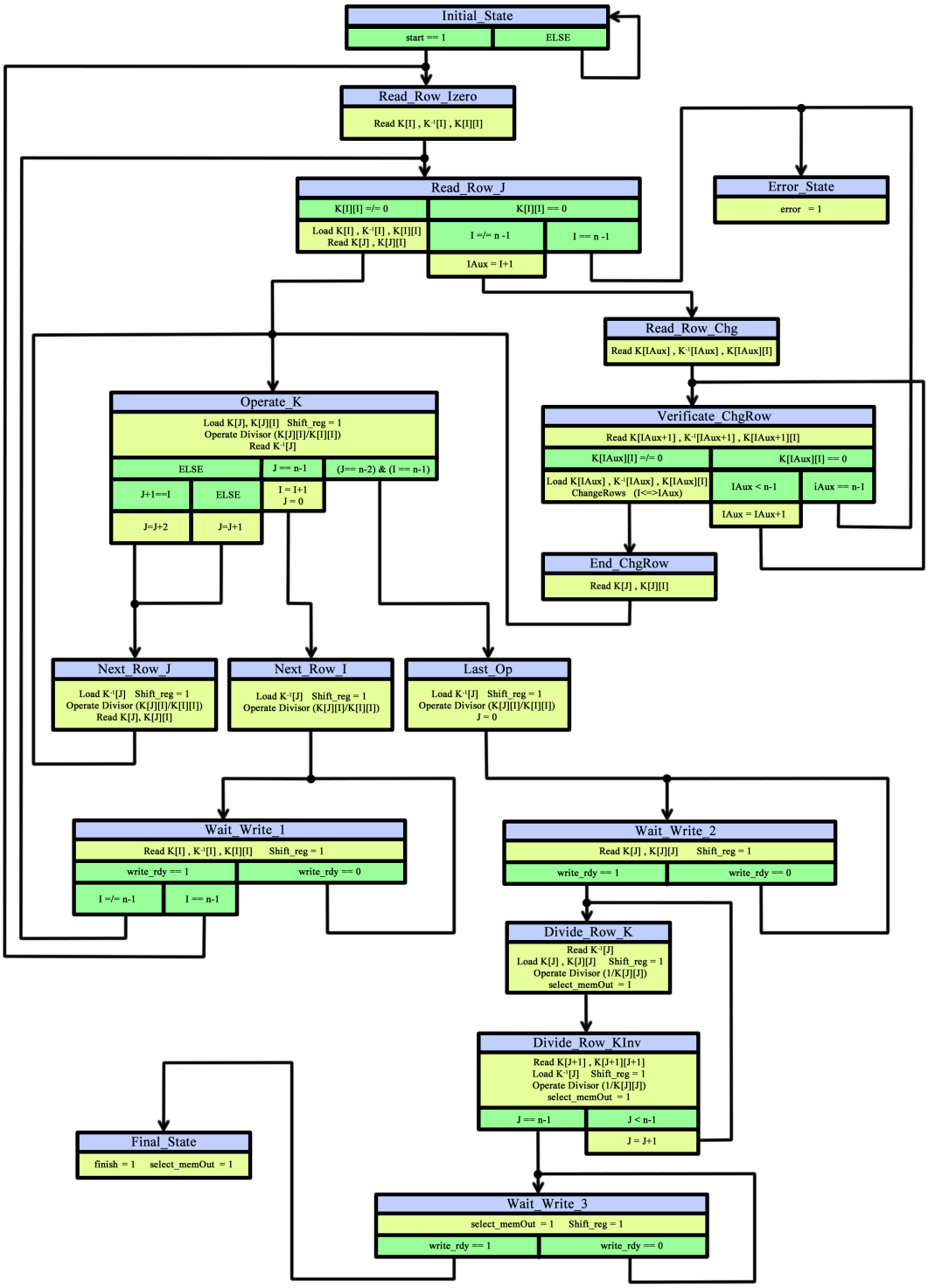
A partir de este estado los resultados de salida de la ruta de datos no vendrán dados por la salida de los restadores, sino por la de los multiplicadores. ya que no se precisa de ninguna resta para formar la matriz identidad a partir de la matriz diagonal.

- ***Divide_Row_KInv:*** *leyendo ($K[J+1], K[J+1][J+1]$), disponible($K^{-1}[J]$).* El estado se encarga de realizar la misma operación en el divisor que el estado predecesor, pero en este caso posteriormente se utiliza el resultado para realizar el cálculo sobre la fila de la matriz

inversa. Una vez activadas las señales correspondientes para la ruta de datos, se contemplan dos posibilidades:

- Si J es menor que n-1. Implica que no se ha llegado todavía a la última fila de la matriz, por lo que se aumenta en una unidad el valor de J, se realiza *prepara($K^{-1}[J]$)* y se vuelve al estado *Divide_Row_K*.
- Si J es igual a n-1. Se ha operado la última fila de la matriz y en consecuencia se han terminado de realizar las iteraciones en el segundo bucle del algoritmo. Se realiza la transición a un estado que espera a la finalización de las escrituras de las filas calculadas (*Wait_Write_3*).

- ***Wait_Write_3:*** Es un estado que espera a la finalización de las escrituras de las filas calculadas por parte del controlador de memoria descrito en el punto anterior (5.2.2.3). Mientras el controlador de memoria no envíe la señal de finalización de las escrituras (que *write_rdy* sea igual a uno) se continuará en el mismo estado. Si la señal de *write_rdy,* en cambio, se activa, se realiza la transición al estado *Final_State*.
- ***Error_State:*** La función de este estado es la de indicar a la unidad de control mayor la imposibilidad de calcular la matriz inversa dada la matriz de entrada. Es un estado *trampa*, para salir de él se debe resetear la máquina de estados.
- ***Final_State:*** La función de este estado es la de indicar a la unidad de control mayor la correcta finalización del proceso. En la memoria K se encontrará la matriz identidad, mientras que en la memoria Kinv se dispondrá de la inversa de la matriz de entrada. Es un estado *trampa*, para salir de él se debe resetear la máquina de estados.



Leyenda

Azul *: nombre de los estados*
Amarillo: *operaciones a realizar (dependen de las condiciones que tengan por encima)*
Verde *: sentencias condicionales (estas sentencias afectan a las operaciones que se encuentren debajo de ellas)*

Figura 5.8: Diagrama de estados del módulo inversa.

5.2.3. Módulo multiplicador matricial

El módulo del multiplicador matricial es el encargado de realizar las distintas operaciones de multiplicación de matrices necesarias para el cálculo de los δ^{RX}. Obtiene los datos para realizar el cálculo de dos memorias: memoria interna K^{-1}, para coger los valores de la matriz inversa calculada mediante el módulo de la inversa, y de una memoria externa, que contiene los distintos valores de $(X - \mu)$, ya calculados, de cada píxel de la imagen.

El módulo cuenta con dos entradas de datos por las que recibe los valores mencionados en el párrafo anterior de las correspondientes memorias. También posee una señal de comienzo (*start*) que indica al módulo cuándo debe comenzar su ejecución, una señal reinicio (*restart*) utilizada para restablecer la máquina de estados del controlador de la multiplicación a su estado inicial de forma síncrona. Cuenta dos señales de salida de datos: la primera indica el resultado obtenido en el cálculo de la actual ejecución (dada por la ruta de datos del multiplicador) y la otra envía el índice del correspondiente píxel dentro de la matriz original (dada por la unidad de control que es la encargada de llevar los índices del cálculo). El módulo tiene una señal de finalización (*finish*) que indica a la unidad de control principal del RX cuándo se ha terminado de realizar el cálculo.

El multiplicador matricial cuenta con tres submódulos (*Figura 5.9*):

- ***vector_N_mult***: corresponde a la **ruta de datos** del módulo. Se encarga de realizar el cálculo de la multiplicación de dos vectores dados. Se encuentra diseñada de esa forma debido a que en el proceso de multiplicar dos matrices las operaciones básicas que se realizan son la multiplicación de vectores (filas por columnas). Tiene dos entradas de datos (*dataA* y *dataB*) por las que obtienen ambos vectores, una señal *opera* que viene de la unidad de control del multiplicador y le indica cuándo puede realizar un cálculo con los datos que tiene en las entradas. También posee dos salidas: una envía el resultado del cálculo de *dataA* * *dataB* y la otra es una señal de *ready* que avisa cuándo puede ser recogido el resultado.
- ***shiftBankNRegs***: banco de registros de desplazamiento para guardar resultados parciales. Es necesario pues se deben realizar dos

multiplicaciones de matrices y por lo tanto guardar el resultado de una de ellas para poder ser utilizado como elemento de cálculo en la otra.

- ***Multiplier_CU***: unidad de control del módulo. Se encarga de controlar los submódulos mencionados anteriormente de forma que su ejecución sea la esperada.

En los siguientes puntos se desarrolla el funcionamiento de cada uno de los submódulos de forma más detallada.

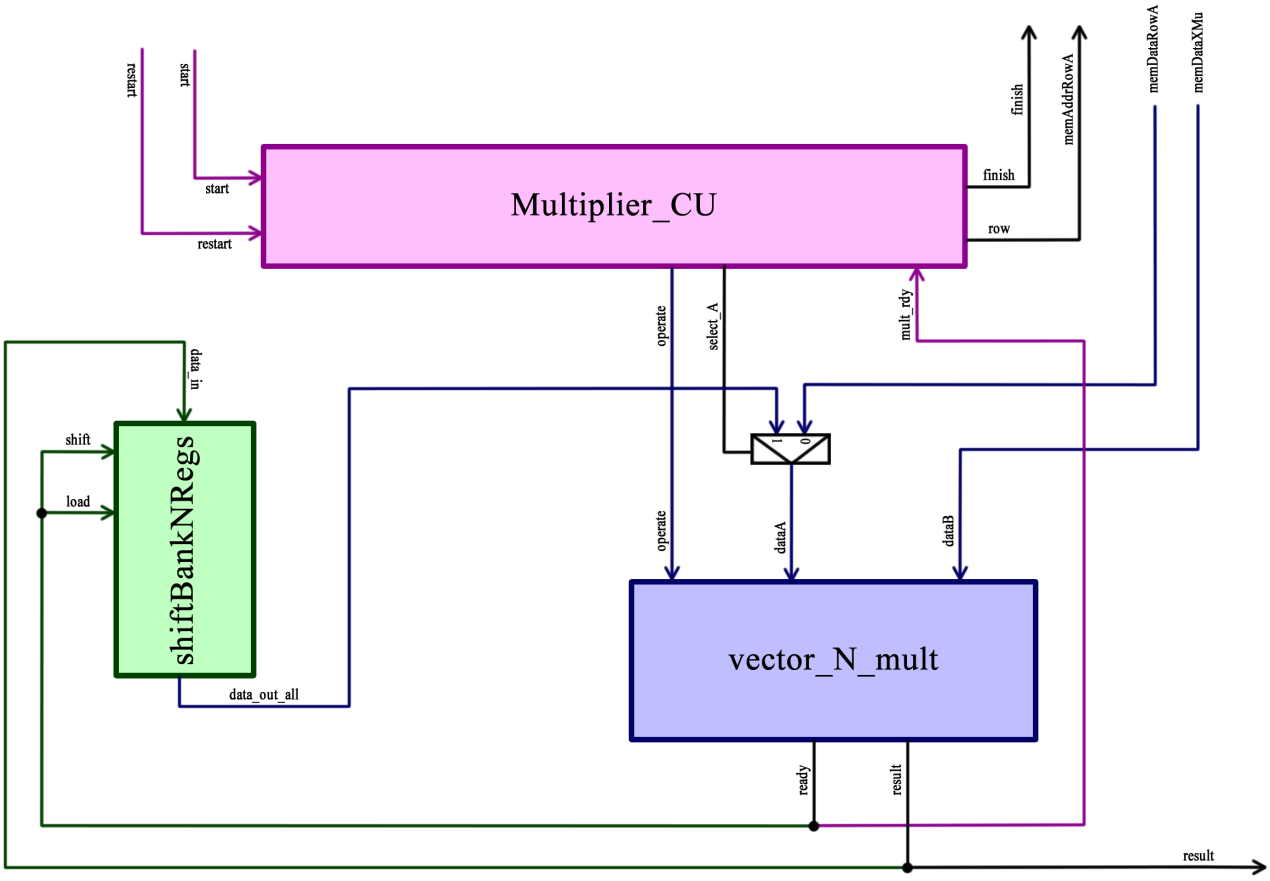


Figura 5.9: Esquema general del módulo del multiplicador matricial.

5.2.3.1. Ruta de datos

Submódulo encargado de calcular el resultado de multiplicar dos vectores de dimensión n. Como se sabe, el cálculo de la multiplicación de dos vectores de dimensión n (a y b) da como resultado un escalar, y se realiza multiplicando cada elemento *i*-ésimo del vector a por su correspondiente *i*-ésimo del vector b. Tras estas multiplicaciones se suman los resultados parciales obtenidos y el valor resultante es el escalar que se busca.

La ruta de datos implementa este concepto con una estructura en forma de árbol (*Figura 5.10*) donde el primer nivel consta de n multiplicadores encargados de realizar las distintas multiplicaciones de cada elemento $a(i) * b(i)$. Tras esto, para agilizar la suma, hemos aprovechado la propiedad asociativa de ésta ($A + (B + C) = (A + B) + C$) mediante la cual se pueden ir haciendo sumas parciales de dos elementos en paralelo hasta obtener el resultado del sumatorio general. El árbol de sumadores tiene una profundidad de $\log_2 n$ siendo n el número de elementos de cada uno de los vectores a multiplicar. El multiplicador está pensado para operar con vectores de un tamaño igual a una potencia de dos. Aunque podría implementarse para que fuese adaptable a cualquier tamaño, se ha dejado así ya que para los casos de pruebas de este proyecto se ha utilizado un tamaño cuatro (2^2) y en la obtención de resultados se ha operado con tamaño treinta y dos (2^5). También ha sido implementado en esta línea para facilitar una estructura modular del submódulo ya que al tratarse de un árbol binario y utilizar potencias de dos, el árbol queda completo y no hay que distinguir casos especiales.

El módulo cuenta con una señal *opera* que entra a cada uno de los multiplicadores del primer nivel por el puerto *operation_nd*, dicha señal indica al módulo cuando tiene los datos de entrada listos y puede usarlos para realizar el cálculo. Los datos de entrada son de un bus con n datos y cada dato es un número en punto flotante de 32 bits, por tanto, la entrada es distribuida de forma uniforme (de 32 en 32 bits) en los distintos multiplicadores. Cada multiplicador cuenta con dos salidas: la primera es el resultado del cálculo y se conecta en el primer nivel de sumadores al sumador y puerto correspondiente, según la estructura de árbol, la otra salida es una señal *ready* que indica cuando el multiplicador ha terminado de realizar el cálculo; estas salidas *ready* se combinan realizando la función lógica *and* de dos en dos (cada par de multiplicadores) y el resultado se conecta al puerto de entrada *operation_nd* de los sumadores del primer nivel, indicándoles cuándo pueden comenzar el cálculo de la primera suma parcial.

Los sumadores cuentan con las mismas entradas y salidas que los multiplicadores, pero realizan la suma en punto flotante en lugar de la multiplicación. En caso de que un sumador tenga más niveles por debajo, las interconexiones entre los del nivel actual y el siguiente se realiza de forma análoga a la de los multiplicadores con los sumadores del primer nivel. De esta forma se construye una estructura en *cascada* en la cual unos módulos avisan a otros de su finalización.

Es el sumador del último nivel es el que obtiene el resultado y su señal *ready* es la que sale del módulo de la ruta de datos para avisar de que el resultado está listo para ser recogido.

Cada multiplicador de la ruta de datos está diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (ipCore). Se han creado con las siguientes propiedades:

- *Latencia*: 1 ciclos
- *Ciclos por operación*: 1 ciclo
- *Puertos de entrada*: 2 (A y B)
- *Puertos de salida*: 1 (resultado)
- *Puertos de control*:
  - *operation_nd*: puerto de entrada que indica al multiplicador cuándo debe operar con los valores que tiene en sus entradas.
  - *ready*: puerto de salida que indica la conclusión de la operación.
- *Operación a realizar*: A * B
- *Precisión de punto flotante*: simple
- *Optimización*: máximo uso (3* DSP48E1)

También cada sumador de la ruta de datos está diseñado mediante las librerías de generación automática de módulos del entorno Xilinx ISE (ipCore). Se han creado con las siguientes propiedades:

- *Latencia*: 1 ciclos
- *Ciclos por operación*: 1 ciclo
- *Puertos de entrada*: 2 (A y B)
- *Puertos de salida*: 1 (resultado)

- *Puertos de control*:
  - *operation_nd*: puerto de entrada que indica al multiplicador cuándo debe operar con los valores que tiene en sus entradas.
  - *ready*: puerto de salida que indica la conclusión de la operación.
- *Operación a realizar*: A + B
- *Precisión de punto flotante*: simple
- *Optimización*: uso completo (2* DSP48E1)

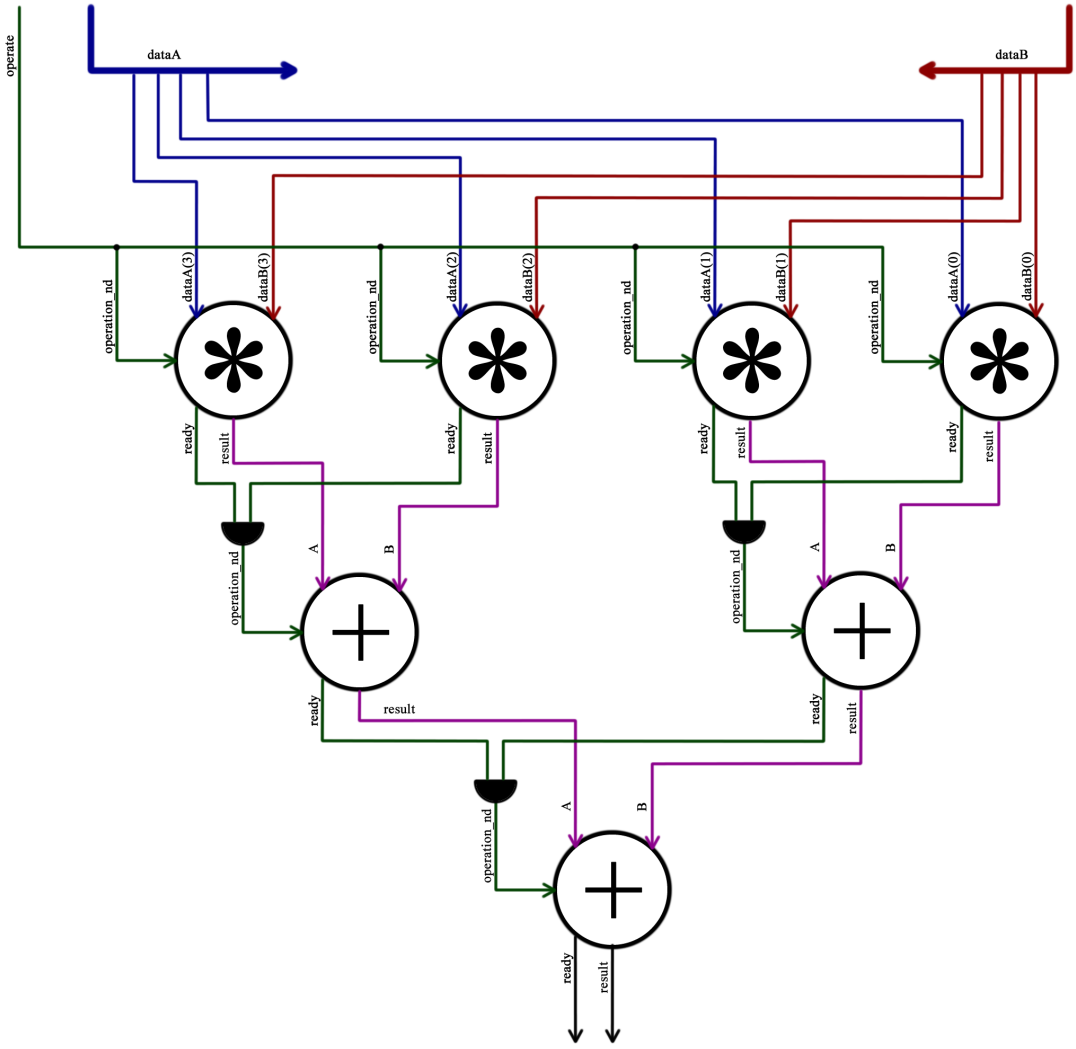


Figura 5.10: Ejemplo de estructura e interconexión de los componentes pertenecientes a la ruta de datos del multiplicador matricial para un tamaño de matriz K 4x4.

5.2.3.2. Banco de registros de desplazamiento

El banco de registros de desplazamiento se utiliza para guardar los resultados parciales de la multiplicación de dos matrices, en este caso y como veremos en la unidad de control, guarda el resultado de la multiplicación $K^{-1} * (x - \mu)$ lo cual nos da como resultado un vector de tamaño n (dimensión de la matriz cuadrada K^{-1}). Este banco de registros obtiene cada resultado del cálculo fila por columna de dicha multiplicación. Realiza desplazamiento de los datos cada vez que llega un nuevo resultado.

Para poder obtener el resultado completo (vector de tamaño n) basta con combinar el valor de todos los registros en una única salida. El orden de guardado de los datos y correcto funcionamiento para el algoritmo se discute en el siguiente punto (unidad de control).

Para poder guardar todos los resultados el banco de registros, éste se crea con n registros cada uno de ellos de 32 bits que guardarán los valores de punto flotante obtenidos durante el cálculo.

El módulo tiene los siguientes puertos de entrada:

- ***data_in***: dato de entrada del módulo. Es de 32 bits, representación en punto flotante. Cuando la señal load está activa y llega el ciclo de reloj, se carga este valor en el primer registro del banco.
- ***load***: señal de carga del banco de registros. Indica cuándo se debe cargar el valor de *data_in* en el primer registro del banco. Esta entrada está conectada al ready de la ruta de datos para que se carguen los valores de su puerto resultado (*result*).
- ***shift***: señal de desplazamiento del banco de registros. Si está activa se realiza el desplazamiento de los registros. Cada registro *i* vuelca su resultado en el *i+1*. El último registro desecha su resultado pues no tiene un registro siguiente donde volcarlo. Esta entrada está conectada al *ready* de la ruta de datos para que se desplace siempre que se obtenga un resultado.

A continuación se exponen los puertos de salida utilizados:

- ***data_out_all***: salida de datos resultado. Combina los valores de todos los registros del banco. Siendo n el número de registros del banco, y cada registro de 32 bits, el ancho de *data_out_all* es de $n * 32$.

Hay que tener en cuenta que al tratarse de un módulo que puede ser instanciado de forma genérica para distintos valores, se trata del mismo módulo utilizado en la ruta de datos del cálculo de la inversa aunque para los valores necesarios para el caso actual, por tanto el módulo cuenta con más salidas de las expuestas previamente, pero no son útiles para los propósitos del multiplicador matricial, por tanto no han sido mencionados.

5.2.3.3. Unidad de control

La unidad de control se encarga de dirigir los submódulos mencionados anteriormente para conseguir computar la multiplicación matricial. En este caso se deben resolver las dos multiplicaciones de matrices que aparecen en el cálculo de δ^{RX}:

$$\delta^{RX}(x) = (x - \mu)^T * \mathbf{K}^{-1} * (x - \mu)$$

Atendiendo a las dimensiones de las matrices tenemos:

- $(x - \mu)^T$ dimensión $1 * n$
- K^{-1} dimensión $n * n$
- $(x - \mu)$ dimensión $n * 1$

Como se sabe, las matrices cumplen la propiedad asociativa para el producto, la cual se establece como:

$$A * (B * C) = (A * B) * C$$

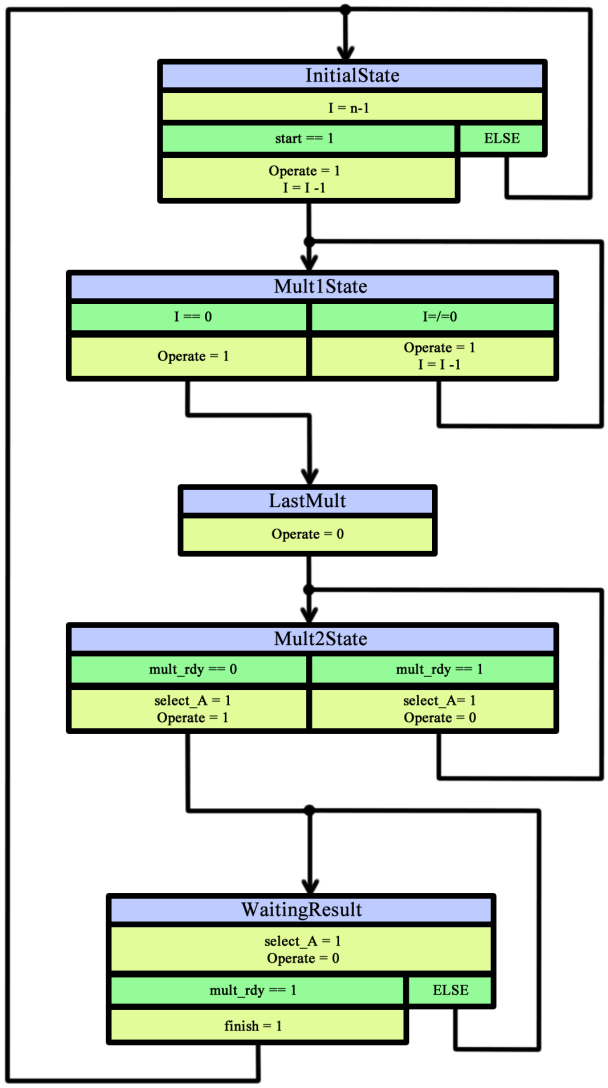
De esta forma se puede realizar primero la multiplicación que se desee siempre que se mantenga el orden de los operandos en las futuras multiplicaciones con respecto al resultado obtenido.

Teniendo en cuenta que se está implementando el algoritmo de multiplicación de matrices en el cual cada paso realiza la multiplicación de un vector fila de la primera matriz por un vector columna de la segunda, se exponen las dos opciones a escoger y cuál es la mejor:

- $\delta^{RX}(x) = [(x-\mu)^T * \mathbf{K}^{-1}] * (x-\mu)$. Si se decide ejecutar primero la multiplicación $[(x-\mu)^T * \mathbf{K}^{-1}]$ se obtiene (atendiendo a las dimensiones anteriores) una matriz resultado de tamaño $1 * n$ habiendo realizado $2n$ multiplicaciones de vectores, a su vez el resultado obtenido debe ser multiplicado por $(x-\mu)$ cuya dimensión es $n * 1$. Esta decisión plantea el problema en este paso, pues la primera matriz (resultado parcial) tiene n columnas y la segunda n filas. Al estar ejecutando el algoritmo de filas por columnas estamos obligados de nuevo a tener que realizar $2n$ multiplicaciones. A parte de esto, cada vector de esta última multiplicación es de tamaño uno, cuestión que habría que tener en cuenta a la hora de realizar el cálculo con la ruta de datos planteada. El coste total de operaciones fila por columna sería $4n$.
- $\delta^{RX}(x) = (x-\mu)^T * [\mathbf{K}^{-1} * (x-\mu)]$. Si se decide ejecutar primero la multiplicación $[\mathbf{K}^{-1} * (x-\mu)]$ se obtiene (atendiendo a las dimensiones anteriores) una matriz resultado de tamaño $n * 1$ habiendo realizado $2n$ multiplicaciones de vectores. En este caso falta realizar la multiplicación de $(x-\mu)^T$por el resultado anterior. $(x-\mu)^T$tiene una dimensión de $1 * n$ y el resultado obtenido $n * 1$. Como se puede ver en este caso no existe problema, pues el primer vector es una fila y el segundo una columna, por tanto la ruta de datos es válida y sólo se debe realizar una única multiplicación para obtener el resultado final. El coste total de operaciones fila por columna con este planteamiento es $2n + 1$.

Como se ha expuesto, el segundo método es mucho más eficiente y plantea menos problemas, es por estas razones que haya sido el elegido para su implementación.

El recorrido de las filas de K^{-1} se debe realizar desde la $n-1$ hasta la 1 ya que de otra forma el vector resultado parcial del banco de registros de desplazamiento no quedaría en el orden correcto según la implementación realizada.



Leyenda

Azul *: nombre de los estados*
Amarillo: *operaciones a realizar (dependen de las condiciones que tengan por encima)*
Verde *: sentencias condicionales (estas sentencias afectan a las operaciones que se encuentren debajo de ellas)*

Figura 5.11: Diagrama de estados de la unidad de control del multiplicador matricial.

La unidad de control se basa en una máquina de estados para realizar la ejecución del algoritmo. Dicha máquina sólo empieza el cálculo con la llegada de una señal de comienzo (*start*), y a su finalización manda una señal de fin (*finish*). Al igual que sucedía en el cálculo de la inversa, se están leyendo datos de memoria, por lo que debemos tener en cuenta los retardos asociados a ésta. Se va a utilizar también un valor I para saber en todo momento qué fila se va a leer de la memoria y también

saber en qué paso se encuentra el algoritmo. A continuación se explica cada uno de los estados (*Figura 5.11*):

- ***InitialState*:** estado inicial de la máquina de estados. Por reinicio el valor de I es $n-1$ (última fila) siendo n una de las dimensiones de la matriz cuadrada K^{-1} en memoria. Por defecto estamos *leyendo*($K^{-1}[I]$). Se permanece en este estado mientras no llegue la señal de comienzo (*start*). Una vez llega dicha señal se manda operar a la ruta de datos en el siguiente ciclo y se decrementa I para *preparar*($K^{-1}[I-1]$).
- ***Mult1State*:** (estado multiplicación 1) estado para terminar el resto de multiplicaciones de $\mathbf{K}^{-1} * (x-\mu)$. En este estado (respecto del valor actual de la I) *tenemos*($I+1$), *leyendo(I)*. Se manda siempre operar para el siguiente estado. Cada vez que se obtiene un resultado en la ruta de datos, éste se guarda en el banco de registros de desplazamiento para su posterior uso. Se diferencian dos casos:
  - Si I es distinto de cero significa que no se han terminado de recorrer las filas, por lo tanto se decrementa la I y se realiza la acción *preparar*($K^{-1}[I-1]$). Se permanece en este estado.
  - Si I es igual a cero significa que se ha llegado a la primera fila y no se tiene que seguir decrementando la I. Se pasa al estado *LastMult* para dar tiempo a la llegada de los datos y realizar las últimas multiplicaciones restantes.
- ***LastMult*:** (última multiplicación) estado para permitir la llegada de los datos y acabar de operar las filas restantes. Se pide a la ruta de datos que deje de operar en el siguiente ciclo y se pasa al estado *Mult2State* para realizar la multiplicación de $(x-\mu)^T$ con el resultado parcial contenido en el banco de registros de desplazamiento.
- ***Mult2State*:** (estado multiplicación 2) estado para realizar la multiplicación del resultado parcial guardado en el banco de registros de desplazamiento con $(x-\mu)^T$. Se cambia el valor del selector del puerto A de la ruta de datos para que obtenga su valor de la salida del banco de registros. En este estado se permanece hasta que la ruta de datos deje de operar (mult_ready de la ruta de datos a cero), esto se realiza para poder dar tiempo a que se almacenen en el banco de registros todos los resultados parciales. Una vez que la ruta de datos ha terminado y mult_ready vale cero, se manda operar en el siguiente estado a la ruta de

datos (para realizar la operación del banco de registros por $(x-\mu)^T$) y se transita al estado *WaitingResult* para proceder a esperar el resultado.

- ***WaitingResult*:** (espera de resultado) se deja de operar la ruta de datos y esperamos a que ésta acabe de realizar los cálculos, por lo tanto mientras *mult_ready* no tome el valor uno no se transita de estado. Una vez la ruta de datos da el resultado de la multiplicación y mult_rdy vale uno, se manda la señal de finalización (*finish*=1) para indicar que ya puede ser recogido el resultado y se transita al estado *IntialState* para esperar posibles nuevas ejecuciones.

5.2.4. Modulo lista ordenada

Es el módulo encargado de ordenar por "nivel de anomalía" cada píxel de la imagen. Tiene una capacidad de almacenamiento de índices de píxeles igual al número de anomalías que se quieren detectar. Aquellos píxeles cuyo nivel de anomalía se encuentre por debajo de los valores ya almacenados de otros píxeles en este módulo serán desechados.

Los datos de entrada se componen de: una señal en la que se especifica la posición del píxel nuevo a tratar dentro de la imagen (*indice_in*), otra señal que contiene el nivel de anomalía previamente calculado en el módulo del multiplicador matricial del píxel en cuestión (*valor_in*) y una señal de carga para realizar la actualización cuando lleguen las entradas anteriores (*load_in*).

La salida está constituida por un bus con un ancho de s x m bits, siendo s el número de anomalías que se quieren detectar en la imagen. Los m bits menos significativos corresponden al índice dentro de la imagen en la que se encuentra el píxel con mayor anomalía, los siguientes m bits a la dirección del segundo píxel con mayor anomalía y así sucesivamente.

Se trata de un módulo basado en una red iterativa en el que cada celda controla la actualización del índice y del valor de la anomalía, por lo que no hay una unidad de control centralizada que dirija sus cálculos (*Figura 5.12*). Al comienzo de su ejecución los registros en los que se almacena el nivel de anomalía de los píxeles se encuentran inicializados a $-\infty$ para que incondicionalmente se guarden los primeros resultados que llegan, aunque siempre correctamente ordenados.

Cada vez que entran los valores de un píxel nuevo al módulo se realiza la comparación del valor de anomalía de este píxel con cada uno de los valores de los píxeles guardados en los registros (todas las comparaciones se realizan en paralelo). Si el valor del nuevo píxel es menor que el valor del i-ésimo registro no se modifica el contenido de los registros. En el caso que sea mayor que el valor del i-ésimo registro nos fijaremos en el valor dado por la (i-1)-ésima comparación. Si en la (i-1)-ésima resulta que el valor del píxel nuevo era menor que el del registro entonces se cargan los datos del nuevo píxel en la posición i-ésima. Si por el contrario resulta que el valor era mayor quiere decir que ya hemos metido los valores del nuevo píxel anteriormente y tenemos que desplazar los restantes, por lo que los nuevos valores de la i-ésima posición serán los de su predecesora.

Si todos los registros contenían valores de píxeles anteriores y el nuevo valor es mayor que alguno de ellos y se debe guardar, el módulo desechará el menor valor que había guardado hasta el momento, quedándose siempre con un máximo de s valores como ya se había mencionado anteriormente.

Para el almacenamiento de los valores de cada píxel se utilizan dos registros (uno para el valor de anomalía del píxel y otro para la posición del píxel dentro de la imagen). La distinción de casos para actualizar los registros, según las operaciones realizadas, se implementa mediante multiplexores.

Tabla de verdad de los multiplexores:

S1 S0	**Salida**
0 0	Conserva valores
0 1	Conserva valores
1 0	Carga los valores de la anomalía entrante al módulo
1 1	Carga los valores del registro anterior

Tabla 5.1: Tabla de verdad de multiplexores lista ordenada.

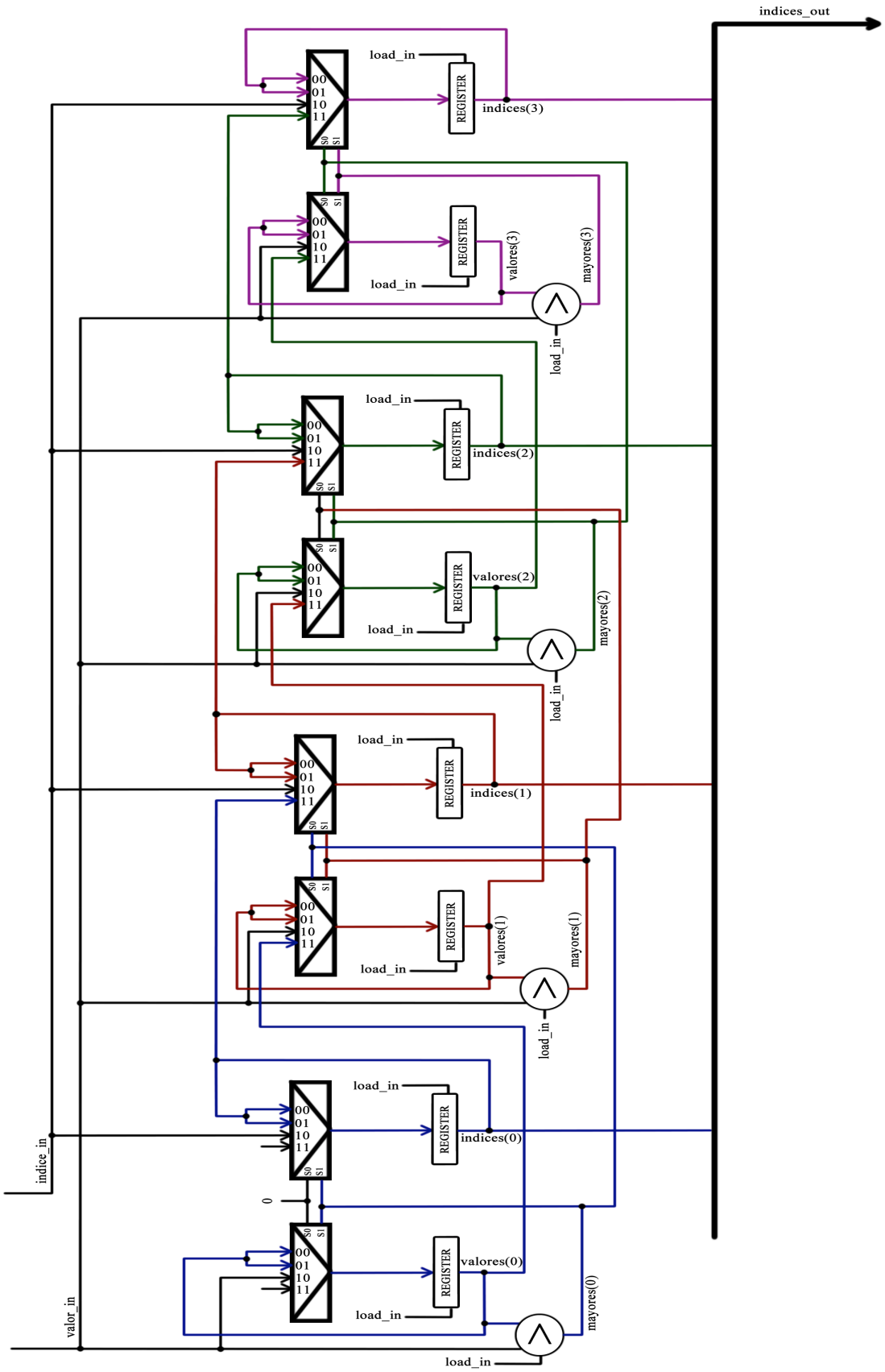


Figura 5.12: Ejemplo de cableado del módulo lista ordenada.
En este caso para almacenar 4 anomalías.

5.2.5. Unidad de control RX

La unidad de control para el cálculo del algoritmo RX es la unidad de control maestra. Se encarga de gestionar los módulos anteriormente citados. Posee un total de 9 estados.

Comienza a realizar los cálculos con la llegada de la señal *start* procedente del exterior.

La comunicación con el módulo de la inversa se realiza mediante el envío de la señal *start_invModule*, que avisa al módulo que esta unidad requiere de su inicio, y las señales *error_invModule* y *finish_invModule*, que avisan a la unidad de control de la finalización correcta o incorrecta del módulo.

Las señales de control que posee para el módulo *Matrix mult* son: start_matrix_Mult, para que comience a realizar los cálculos; *restart_matrix_Mult,* que prepara el módulo para el comienzo de otra ejecución; finish_matrix_Mult, que avisa a la unidad de control de la finalización del módulo.

Otras señales que tiene la unidad de control son: *addrRow_XMu_Ext,* la cual solicita filas de la memoria $x - \mu$ externa para que posteriormente lleguen los datos al módulo MatrixMult; *addrRowK_Ext,* para pedir las filas de la memoria exterior K y poder inicializar la memorias K interior con los datos que lleguen; *addrRowKInt,* que direcciona las filas de las memorias internas para ir guardando los datos que lleguen en el caso de la matriz K, o formar la matriz identidad en el caso de K^{-1}; *wrRow_Mem,* señal de escritura utilizada para la inicialización de las matrices; *dataRow_KInv,* datos para la inicialización de la matriz identidad en la memoria K^{-1}; *selectDataIn_Mem,* que selecciona qué módulo puede leer de la memoria.

Tabla de verdad de *selectDataIn_Mem*:

S1 S0	Módulo Asignado para las lecturas o escrituras
0 0	Unidad de Control (inicialización)
0 1	Inv Module
1 0	Matrix mult
1 1	Matrix mult

Tabla 5.2: Tabla de verdad de módulo asignado para lecturas o escrituras lista ordenada.

La unidad de control cuando se resetea, ya sea síncrona o asíncronamente, inicializa los valores de las señales de dirección de memorias (*addrRowK_Ext, addrRowKInt, addrRow_XMu_Ext*) a cero y asigna como estado inicial de la máquina de estados a *Wait_Start.* Además, el valor *addrRowK_Ext* se manda por defecto a las memoria exterior para su lectura al comienzo, por lo que cuando llega el momento de ejecutar el algoritmo sólo habrá que esperar un ciclo de retardo (*leyendo*).

A continuación se explica detalladamente cada uno de los estados de la unidad de control (*Figura 5.13*):

- ***Wait_Start:*** *leyendo (*$KExt[0]$*)*. Es el estado inicial. Se encarga de realizar la espera hasta la llegada de la señal de comienzo para iniciar el cálculo. Si esta señal es activada se incrementa addrRowK_Ext en una unidad, se realiza la acción *preparar* $KExt[addrRowK\_Ext]$ y se produce la transición al estado *Initialization*.

  El incremento de addrRowK_Ext en una unidad es de suma importancia pues el retardo producido por la lectura de la memoria K externa haría que en el siguiente estado se estuviera escribiendo en la fila i-ésima en K y la i+1-ésima en K inversa. De este modo conseguimos que escriba la misma fila cada vez que hay una escritura.

- ***Initialization:*** *disponible (* $KExt[addrRowK\_Ext - 1]$ *), leyendo* ($KExt[addrRowK\_Ext]$). La funcionalidad de este estado consiste en inicializar las memorias interiores, una con la matriz K y otra con la matriz identidad. Como se ha podido deducir de lo descrito en el estado anterior el valor de $addrRowK\_Ext$ siempre será una unidad mayor que

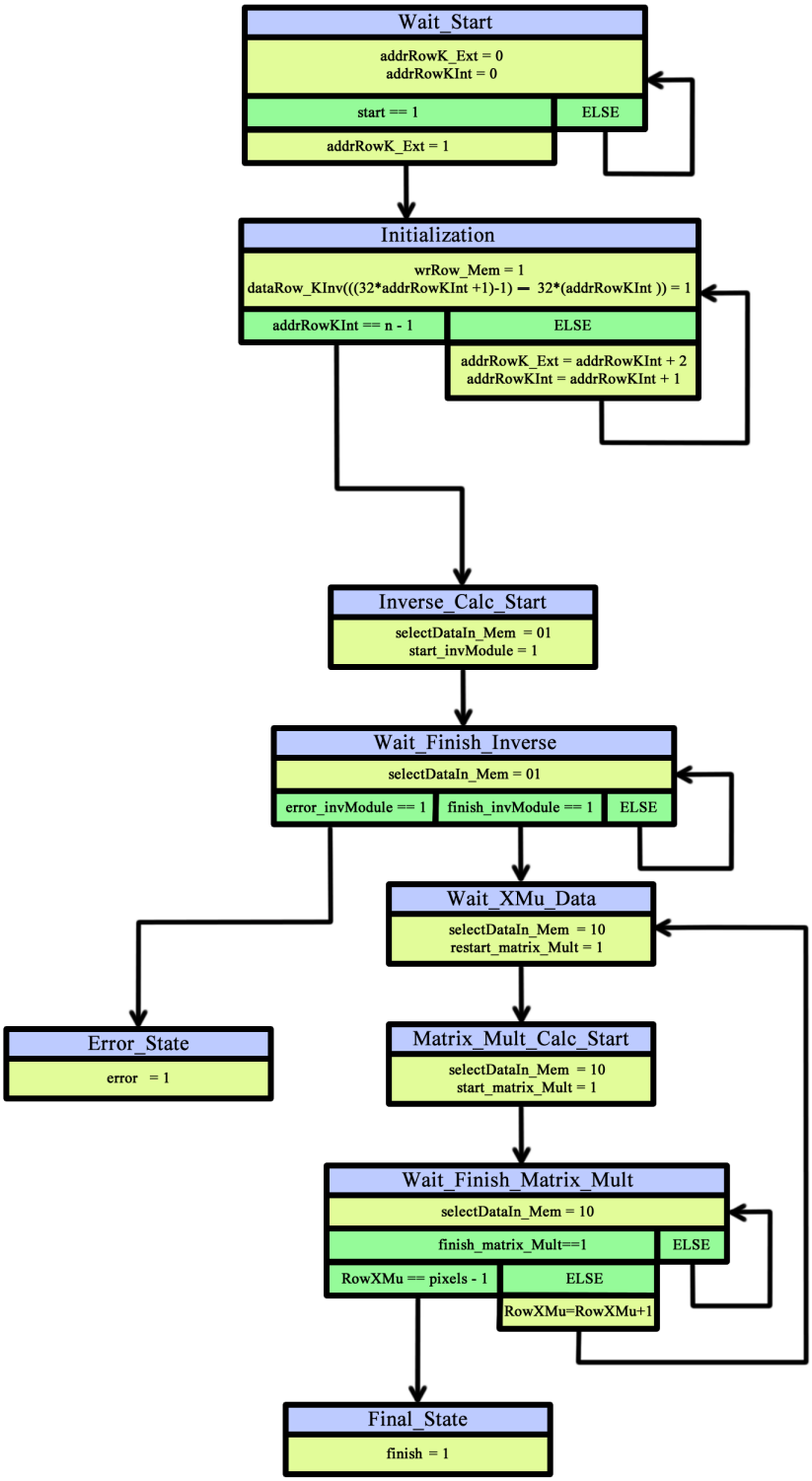
addrRowKInt para compensar el retardo de la memoria y poder escribir las mismas filas.

En el estado se realiza la escritura de la fila addrRowK_Ext-1 en la matriz K y de la fila *addrRowKInt* (que contiene la fila I[*addrRowKInt*] de la matriz identidad). Una vez activadas las señales correspondientes, se contemplan dos posibilidades:

- addrRowKInt es menor que n-1. Se incrementan addrRowKInt y addrRowK_Ext y se realiza preparar $KExt[addrRowK\_Ext]$.
- addrRowKInt es igual a n-1. Hemos terminado de inicializar las matrices por lo que realizamos la transición hacia el estado Inverse_Calc_Start.

- ***Inverse_Calc_Start:*** Es el estado en el que se avisa al módulo de la inversa para que comience su ejecución. Se activa la señal *start_invModule* y se modifica el valor de la señal *selectDataIn_Mem* para que el módulo de la inversa pueda leer y escribir en las memorias internas. Se produce después la transición al estado *wait_Finish_Inverse.*
- ***Wait_Finish_Inverse:*** Se encarga de realizar la espera hasta que el módulo que calcula la matriz inversa termina. Como ya se había comentado en este módulo, se puede dar la posibilidad de que no exista la matriz inversa para la matriz K dada y que se active la señal de *error_invModule,* en cuyo caso se hará una transición hacia el estado de error *Error_State.* Si el módulo matriz inversa termina satisfactoriamente (se activa *finish_invModule*) se realizará la transición al estado *Wait_XMu_Data,* y darán comienzo las multiplicaciones de matrices.
- ***Wait_XMu_Data:*** Debido a cómo se inicializa la máquina de estados, durante los anteriores estados se producía siempre la lectura de la fila cero de la matriz $x - \mu$externa, ya que *addrRow_XMu_Ext* vale 0 . Por lo que con la llegada a este estado se estará en la acción *leyendo* ($Kx\mu\_Ext[addrRow\_XMu\_Ext\,]$). Se modifica el valor de la señal *selectDataIn_Mem* para que el módulo *Matrix Mult* pueda leer de la memoria K inversa interna y se resetea ese mismo módulo para su posterior uso en el siguiente ciclo. Por último, se realiza la transición al estado *Matrix_Mult_Calc_Start,* que iniciará los cálculos del módulo *Matrix Mult*.

- ***Matrix_Mult_Calc_Start:*** *disponible* ($Kx\mu\_Ext[addrRow\_XMu\_Ext]$). Este estado de la unidad de control es el encargado de enviar la señal de comienzo al módulo *Matrix Mult* (*start_matrix_Mult* se activa a uno). Cabe mencionar que el módulo actuará correctamente al enviarse en el mismo ciclo la señal de comienzo junto con los datos de la matriz $Kx\mu\_Ext$. Se realiza la transición al estado *Wait_Finish_Matrix_Mult* que realiza una espera hasta la finalización del módulo.
- ***Wait_Finish_Matrix_Mult:*** Se encarga de realizar la espera hasta que el módulo *Matrix Mult* termina termina de realizar los cálculos y envía la señal *finish_matrix_Mult*. se contemplan dos posibilidades:
  - addrRow_XMu_Ext es menor que el número de píxeles de la imagen (*pixels*-1). Se incrementa en una unidad $addrRow\_XMu\_Ext$ y se realiza la acción *preparar* ($Kx\mu\_Ext[addrRow\_XMu\_Ext]$). Una vez hecho esto se produce la transición al estado *Wait_XMu_Data* para volver a realizar los cálculos con la siguiente fila de la matriz $Kx\mu\_Ext$.
  - addrRow_XMu_Ext es igual a n-1. Se ha terminado de recorrer la matriz $Kx\mu_Ext$y de realizar los cálculos con sus filas. Se efectúa una transición a un estado de fin, compensando el ciclo de retardo necesario para ordenar el nuevo elemento producido por *Matrix Mult* que obtiene el módulo *lista ordenada.*
- ***Error_State:*** La función de este estado es la de indicar la imposibilidad de calcular la matriz inversa dada la matriz de entrada. Es un estado trampa, para salir de él se debe resetear el módulo RX.
- ***Final_State:*** Estado encargado de avisar de la correcta finalización del algoritmo RX. La señal *index_out* procedente del módulo *lista ordenada* y que se envía al exterior del módulo general contendrá las direcciones de los píxeles con mayores índices de anomalía ordenados por los mayor a menor comenzando por los bits menos significativos. Es un estado trampa, para salir de él se debe resetear el módulo RX.



Leyenda

Azul *: nombre de los estados*
Amarillo: *operaciones a realizar (dependen de las condiciones que tengan por encima)*
Verde *: sentencias condicionales (estas sentencias afectan a las operaciones que se encuentren debajo de ellas)*

Figura 5.13: Diagrama de estados de la unidad de control del módulo RX.

Capítulo 6

Resultados experimentales

6.1. Plataforma reconfigurable

La arquitectura hardware descrita en la sección 5 ha sido desarrollada utilizando el lenguaje VHDL para la especificación del algoritmo. Además, se ha empleado el entorno Xilinx ISE para especificar el sistema completo.

Este sistema ha sido simulado sobre la FPGA Virtex-5QV XC5VFX130T. El uso de esta FPGA se debe a su extendido uso, siendo una de las más actuales en el mercado. También, se basa en las mismas arquitecturas que otras FPGAs endurecidas para radiación [48] que han sido certificadas por diversos organismos internacionales para operar en condiciones espaciales. En concreto, la FPGA Virtex-5QV XC5VFX130T es muy parecida a la FPGA Virtex-5QV XQR5VFX130 certificada para el espacio, por lo que el diseño propuesto se puede implementar en ella de manera inmediata. Las propiedades de esta FPGA adaptada para el espacio se pueden observar en la *Tabla 6.1*.

Voltaje	1.0v
Slices	20,480
Celdas lógicas	130,000
CLB Flip-Flops	81,920
Distribución de memoria RAM máxima	1,580
Bloques RAM/FIFO w/ECC (36Kb cada uno)	298
Bloques RAM/FIFO (18Kb cada uno)	596
Bloques totales de memoria RAM (Kb)	10,728
Máximo número de puertos entrada/salida individuales	836
Máximo número de puertos entrada/salida emparejados	414
Impedancia controlada digitalmente	Si
DSP slices (DSP48E1)	320

Tabla 6.1 : propiedades de la FPGA Virtex-5QV XQR5VFX130.

6.2. Conjunto de imágenes hiperespectrales

Para el presente trabajo de fin de carrera se han empleado dos imágenes hiperespectrales reales, una de ellas tomada por el sensor AVIRIS (desarrollado por el *Jet Propulsion Laboratory* de la NASA) y la otra tomada por el sensor HYDICE (patrocinado por el *U.S. Navy Space and Warfare Systems Command*). Ambas imágenes son utilizadas frecuentemente en aplicaciones de detección de anomalías como referentes. En los siguientes apartados se describen las imágenes hiperespectrales consideradas.

6.2.1. AVIRIS WTC

La primera imagen utilizada en el proyecto fue adquirida por el sensor AVIRIS el día 16 de septiembre de 2001, justo cinco días después de los ataques terroristas que derrumbaron las dos torres principales y otros edificios del complejo *World Trade Center* (WTC) en la ciudad de Nueva York. Las características principales de la imagen aparecen resumidas en la *Tabla 6.2*. Conviene destacar que

la resolución espacial de la imagen es muy elevada (1,7 metros por píxel) para lo que suele ser habitual en el caso de AVIRIS, en el que la resolución espacial habitual suele rondar los 20 metros por píxel. Esto se debe a que la imagen corresponde a un vuelo de baja altura, mediante el cual se pretendía obtener la mayor resolución espacial posible sobre la zona de estudio al contrario que otros estudios con el mismo sensor, en los que se pretende cubrir un área más extensa.

Líneas	614
Muestras	512
Bandas	224
Rango Espectral	0,4 – 2,5 μm
Resolución Espacial	1,7 metros/píxel
Tamaño	140 MBytes aproximadamente

Tabla 6.2: Características de la imagen hiperespectral AVIRIS obtenida sobre la zona del World Trade Center en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de septiembre de 2001.

La *Figura 6.1* muestra una composición en falso color de la imagen AVIRIS WTC. En dicha composición, se han utilizado las bandas espectrales localizadas en 1682, 1107 y 655 nanómetros como rojo, verde y azul, respectivamente. En función de la composición en falso color utilizada, las zonas de la imagen con predominancia de vegetación reciben tonalidades verdes, mientras que las zonas con fuegos aparecen con tonos rojos.

El humo que proviene del área del WTC (enmarcada en el rectángulo de color rojo) y que se dirige al sur de la isla de Manhattan recibe un tono azul claro en la composición de falso color, debido a la elevada reflectancia del humo en la longitud de onda correspondiente a 655 nanómetros.

Figura 6.1: Composición en falso color de la imagen hiperespectral AVIRIS obtenida sobre la zona del WTC en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de septiembre de 2001. El recuadro en rojo marca la zona donde se sitúa el WTC en la imagen.

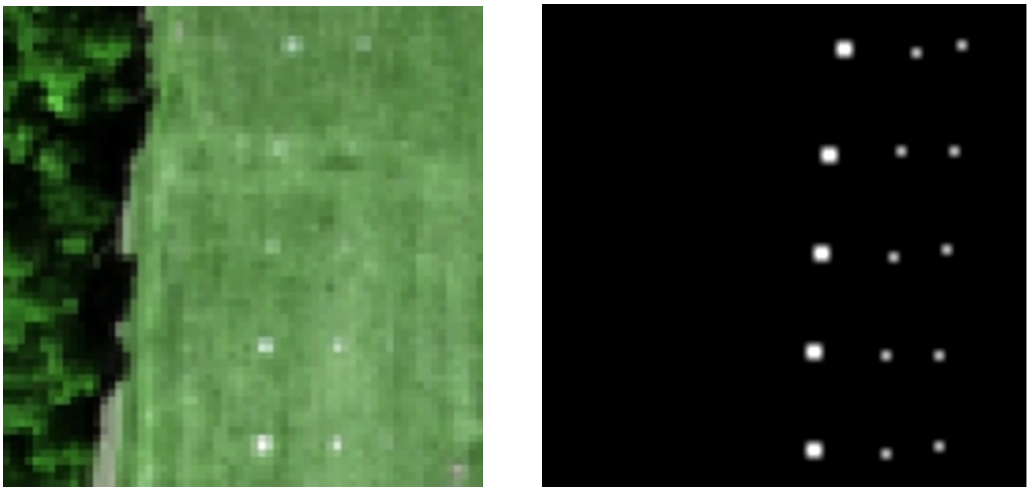
6.2.2. HYDICE

El segundo conjunto de datos hiperespectrales reales fueron recopilados en el marco del *HYperspectral Digital Image Collection Experiment* (HYDICE). La Tabla 6.3 resume las características principales de la imagen. Esta imagen tiene unas dimensiones de 64 x 64 píxeles sobre la que se han colocado 15 paneles de distintos tamaños en la escena. Se dispone del mapa verdad-terreno para la escena, que indica la ubicación espacial de los paneles. Los tamaños de los paneles en la primera, segunda, y tercera columna tienen un tamaño (en metros) de: 3 x 3, 2 x 2 y 1 x 1, respectivamente. La imagen adquirida se compone de 210 bandas espectrales que

cubren un espectro de van de los 0,4 a 2,5 micras. Las bandas con baja relación señal/ruido: 1-3 y 202-210, y las bandas de absorción de vapor de agua: 101-112 y 137-153, fueron eliminadas antes de realizar los resultados experimentales, por lo que finalmente se utilizaron un total de 169 bandas. La resolución espacial de la escena es de 1,56 metros (es decir, la última columna de objetivos son de tamaño sub-píxel) y la resolución espectral es 10 nanómetros. La Figura 6.2 representa la escena HYDICE en composición de falso color.

Líneas	64
Muestras	64
Bandas	210
Rango Espectral	0,4 − 2,5 μm
Resolución Espacial	1,56 metros/píxel
Tamaño	6 MBytes aproximadamente

Tabla 6.3: Características de la imagen hiperespectral HYDICE compuesta por 15 paneles puestos sobre la escena.



(A) *(B)*

Figura 6.2. (A) Representación en falso color de la escena hiperespectral HYDICE y (B) mapa verdad-terreno para la escena.

6.3. Evaluación de las anomalías detectadas

En este apartado se evalúa la precisión de las anomalías detectadas por la implementación propuesta del algoritmo RX utilizando las imágenes hiperespectrales reales descritas anteriormente. Antes de comenzar, se destaca que la implementación descrita en el apartado 5 de esta memoria proporciona exactamente los mismos resultados que una versión software equivalente, por lo que se puede dar por válida la implementación hardware propuesta.

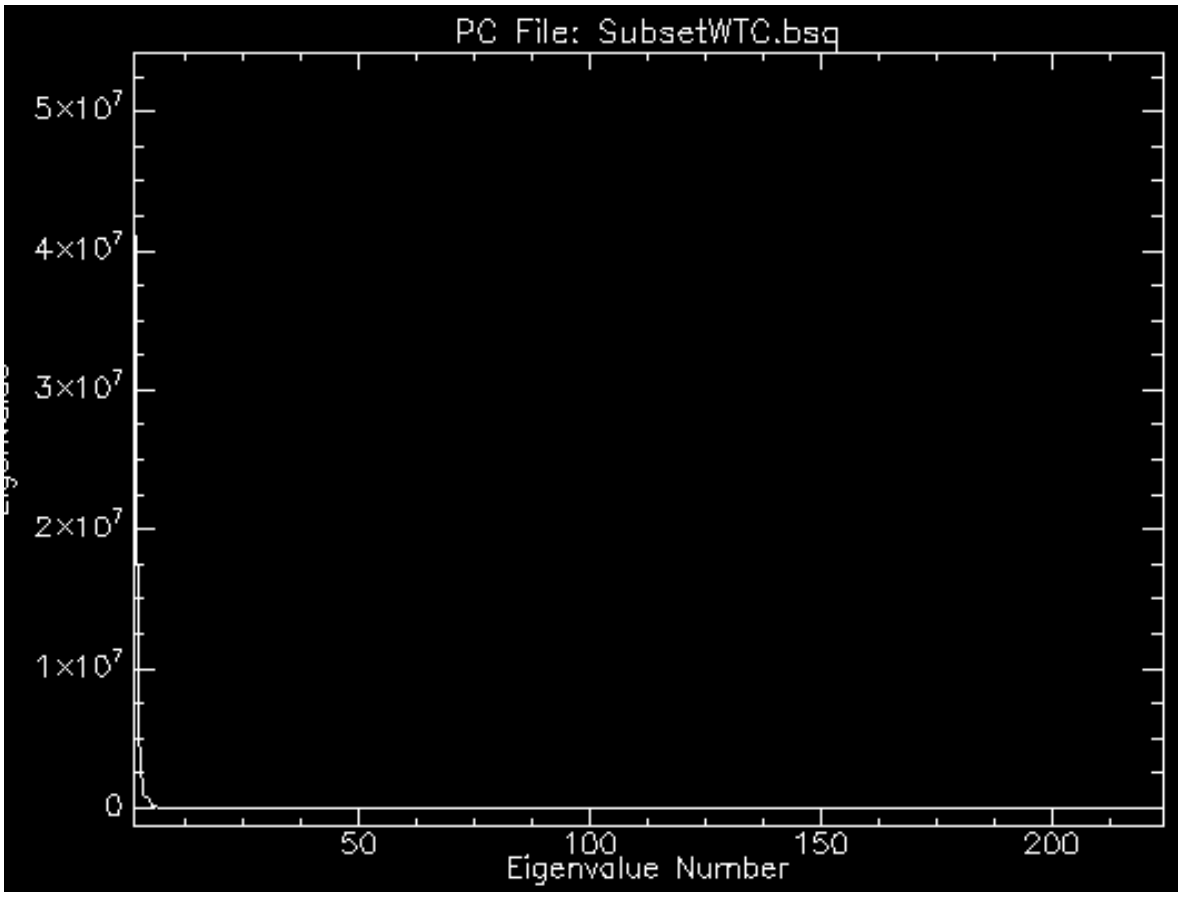


Figura 6.3: Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen AVIRIS WTC.

Primeramente, se ha partido de las imágenes hiperespectrales descritas en el apartado anterior para posteriormente someterlas a un proceso de reducción dimensional. Frecuentemente los píxeles de una imagen hiperespectral se ubican en un subespacio muy pequeño en comparación con el número de bandas disponibles. La identificación de este subespacio permite una correcta reducción de la dimensionalidad, que se traduce en una mejora en el rendimiento, en la complejidad

de los algoritmos y en el almacenamiento de datos. En nuestro caso concreto hemos aplicado el algoritmo *Principal Component Analysis* (PCA) disponible en el software comercial *Environment for Visualizing Images* (ENVI). Las *Figuras 6.3 y 6.4* muestran los autovalores para cada una de las bandas de la nueva imagen tras el análisis de componentes. Como cabría esperar, las primeras bandas son las que acumulan la mayor parte de la variabilidad de la imagen con un acentuado decremento. Finalmente, nuestras imágenes reducidas dimensionalmente constarán de 32 bandas, reteniendo un 99.99% de la información en el caso de la imagen AVIRIS WTC y de un 99.98% en el caso de la imagen HYDICE.

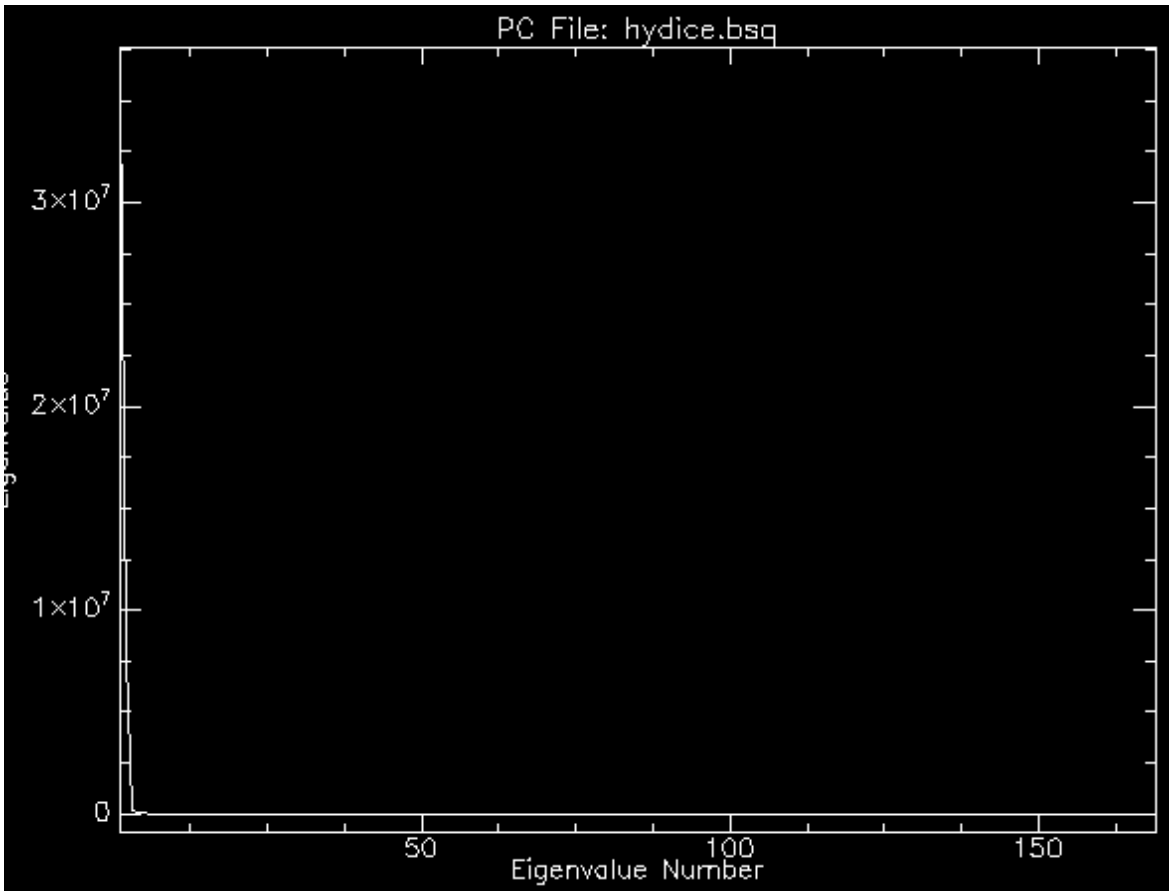
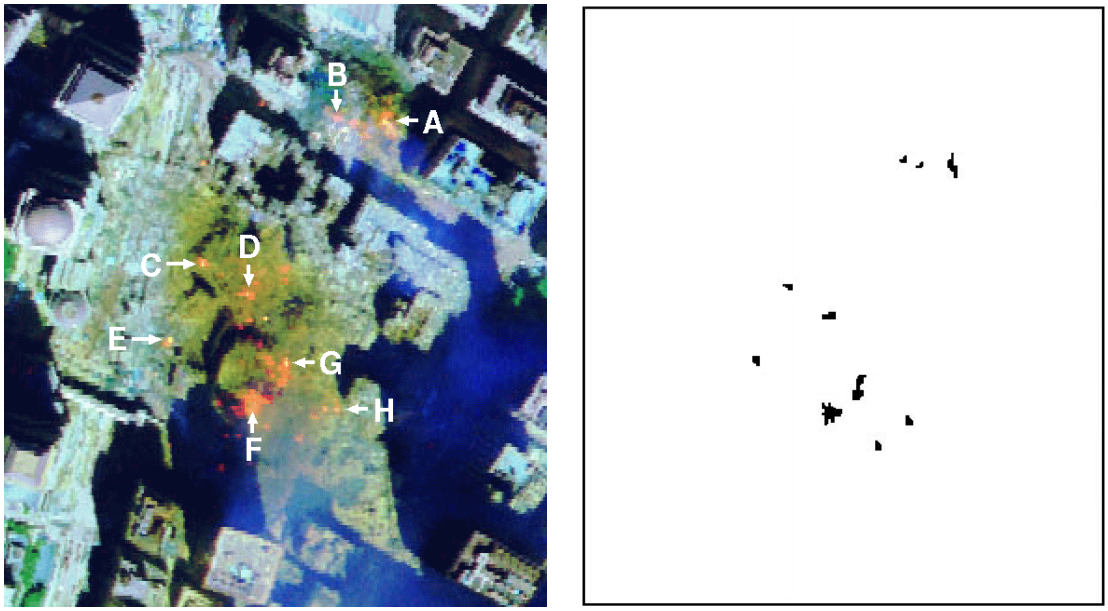


Figura 6.4: Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen HYDICE.

Una vez se han obtenido las imágenes reducidas dimensionalmente, calculamos para cada una su matriz de covarianzas, que servirán de entrada a la implementación del algoritmo RX propuesto. El motivo de no realizar un procesamiento hardware del cálculo de la matriz de covarianzas se debe a la enorme cantidad de entrada/salida que sería necesaria, lo que haría que se produjera una

excesiva penalización y se obtuviera un tiempo de cálculo superior al que se conseguiría con un procesador empotrado. Actualmente, son muy frecuentes las FPGAs con recursos heterogéneos que, entre otros componentes, incluyen procesadores, por lo que se puede realizar un codiseño hardware/software en el que la matriz de covarianzas se calculase de manera software y el resto en hardware.

La precisión del algoritmo va a ser evaluada por su capacidad de detectar automáticamente los puntos calientes del incendio en la imagen AVIRIS WTC y de los paneles en la imagen HYDICE. En ambos casos el número de píxeles anómalos a detectar se ha fijado en 30 después de calcular la dimensionalidad virtual de los datos.



(A) *(B)*

Figura 6.5: (A) Representación en falso color de la escena hiperespectral WTC y (B) su información de realidad sobre el terreno asociado.

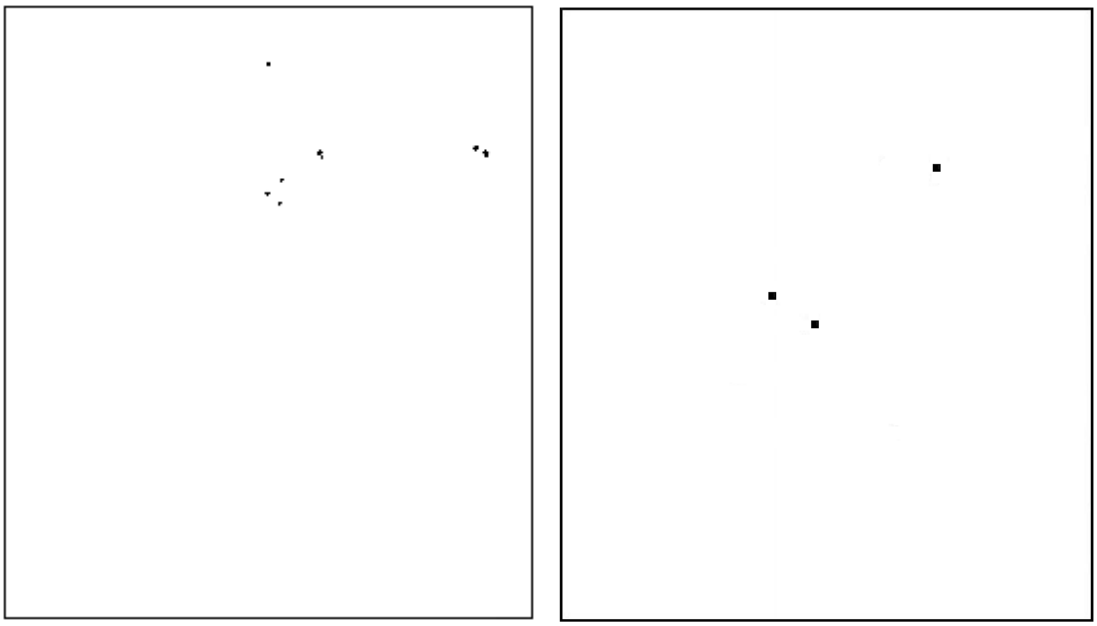
Existe una extensa información de referencia, recopilada por el *U.S. Geological Survey* (USGS), sobre la escena AVIRIS WTC [49]. En este proyecto de fin de carrera, utilizamos el mapa termal del USGS [50] donde se muestra la localización de los puntos calientes (que pueden considerarse anomalías) en el área del WTC, mostrados en rojo brillante, naranja y amarillo en la *Figura 6.5(A)*. El mapa está centrado en la región donde cayeron las dos torres y en el rango de temperaturas que va desde los 700ºF a los 1300ºF (desde los 371ºC a los 704ºC). La *Tabla 6.4* muestra información adicional disponible por el USGS sobre los puntos calientes

(incluyendo localización y temperatura). Dicho mapa termal es el que utilizamos para obtener la información sobre el terreno (ver *Figura 6.5(B))* que serán los puntos que utilicemos para evaluar la correcta detección de anomalías.

Punto caliente	Latitud (Norte)	Longitud (Este)	Temperatura (Kelvin)
'A'	40^{o} 42' 47.18"	74^{o} 00' 41.43"	1000
'B'	40^{o} 42' 47.14"	74^{o} 00' 43.53"	830
'C'	40^{o} 42' 42.89"	74^{o} 00' 48.88"	900
'D'	40^{o} 42' 41.99"	74^{o} 00' 46.94"	790
'E'	40^{o} 42' 40.58"	74^{o} 00' 50.15"	710
'F'	40^{o} 42' 38.74"	74^{o} 00' 46.70"	700
'G'	40^{o} 42' 39.94"	74^{o} 00' 45.37"	1020
'H'	40^{o} 42' 38.60"	74^{o} 00' 43.51"	820

Tabla 6.4. Propiedades de los puntos calientes etiquetados en la Figura 6.5(a).

En el caso de la escena AVIRIS WTC la *Figura 6.6 (A)* muestra los 30 píxeles anómalos detectados en el total de la imagen. Centrándonos en la zona de interés, la *Figura 6.6 (B)* muestra las anomalías detectadas donde se sitúa el WTC. En este caso, sólo tres de las ocho anomalías (es decir, aquellas etiquetadas como 'A', 'C' y 'D') fueron detectadas, lo que coincide con los resultados anteriores publicados en la literatura [51]. Sin embargo, aumentando el número de anomalías a detectar se mejoran los resultados de detección. En este trabajo, hemos decidido detectar 30 anomalías basándonos en el cálculo de la dimensionalidad virtual de los datos, que nos proporciona un criterio objetivo para establecer el número de anomalías. Cabe destacar que si quisiéramos obtener un número mayor de anomalías, tan sólo deberíamos definir un mayor tamaño para el módulo mayores sin que ello produzca penalizaciones en el tiempo de ejecución.



(A) *(B)*

Figura 6.6: (A) Píxeles anómalos detectados por la implementación propuesta en el total de la imagen AVIRIS WTC y (B) en la zona del WTC.

El mismo experimento se realizó para la imagen HYDICE. La *Figura 6.7* muestra las anomalías detectadas por la implementación hardware propuesta. Si la comparamos con la *Figura 6.2 (B)* observamos que todos los paneles de tamaño 3 x 3 fueron detectados como anomalías, todos a excepción uno de los paneles de 2 x 2 y ninguno de los paneles de 1 x 1. Al igual que en la imagen AVIRIS WTC, aumentando el número de anomalías a detectar se mejoran los resultados de detección, aunque basándonos en la dimensionalidad virtual de los datos, hemos decidido detectar solo 30 anomalías. Estos resultados se corresponden con los anteriormente publicados en la literatura [52].

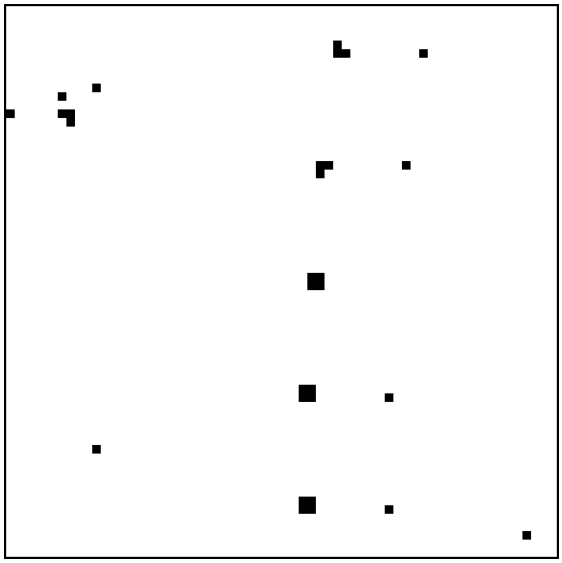


Figura 6.7: Píxeles anómalos detectados por la implementación propuesta en el total de la imagen HYDICE.

6.4. Evaluación del rendimiento

La *Tabla 6.5* muestra los recursos empleados en la implementación hardware propuesta del algoritmo RX para números de bandas diferentes para la FPGA Virtex-5QV XC5VFX130T. Estas FPGA tiene un total de 20480 slices y dispone de numerosos recursos heterogéneos. Un ejemplo de ellos son los Block RAMs (bloques de memoria), de los que se vale el proyecto para implementar las memorias de una forma más eficiente.

Frecuentemente la E/S es el cuello de botella en los sistemas paralelos. Para reducir el número de peticiones externas se ha decidido en esta implementación copiar la matriz de covarianzas en una memoria interna al comienzo de la ejecución para no tener que acceder a ella posteriormente.

Como vemos en la *Tabla 6.5*, con el aumento progresivo del número de bandas de la imagen el consumo de recursos de la FPGA se incrementa, llegando a ser muy ajustado para los casos de 32 bandas (véase los DSP48E1s). Aunque el número de DSP48E1s es casi completo para las imágenes de 32 bandas, si se ejecuta el algoritmo con una imagen con mayor número de bandas, los ipCores que no puedan ser implementados mediante DSP48E1s podrían sustituirse por módulos análogos implementados mediante los *slices*, consiguiendo de esta manera utilizar recursos todavía disponibles sin exceder el número de DSP48E1s máximo de la FPGA.

	Pruebas 4 bandas 16 px	**AVIRIS 32 bandas 614x512 px**	**HYDICE 32 bandas 64x64 px**
Número de registros slice	1.900 (2%)	13.606 (16%)	13.389 (16%)
Número slice LUTs	4.602 (5%)	32.741 (39%)	32.517 (39%)
Número de pares entereos usados LUT-FF	935 (16%)	6.829 (17%)	6.613 (16%)
Número de bloques RAM/FIFO	8 (2%)	64 (21%)	64 (21%)
Número de BUFG/BUFGCTRLs	1 (3%)	2 (6%)	2 (6%)
Número de DSP48E1s	38 (11%)	318 (99%)	318 (99%)

Tabla 6.5: Resumen de los recursos utilizados para la implementación del algoritmo RX en la FPGA Virtex-5QV XC5VFX130T.

En la Tabla 6.6 se muestran las especificaciones de tiempo y frecuencia obtenidas por Xilinx en las implementaciones realizadas.

	Pruebas 4 bandas 16 píxeles	**AVIRIS 32 bandas 614x512 píxeles**	**HYDICE 32 bandas 64x64 píxeles**
Período mínimo	14.555 ns	15.466 ns	15.466 ns
Máxima frecuencia	68.704 MHz	64.656 MHz	64.656 MHz
Mínimo tiempo de la llegada requerido antes del reloj	7.929 ns	7.929 ns	7.929 ns
Máximo tiempo de salida requerido después del reloj	2.814 ns	2.867 ns	2.864 ns

Tabla 6.6: Resumen de tiempos y frecuencia de la implementación del algoritmo RX en la FPGA Virtex-5QV XC5VFX130T.

Como vemos en la *Tabla 6.7*, el tiempo de ejecución se reduce notablemente. Los resultados dan información de los tiempos de procesamiento de la implementación en FPGA considerada y la versión software equivalente desarrollada en lenguaje C y ejecutada en un PC con un procesador AMD Athlon 2.6 GHz y 512 Mb de RAM. En ambos casos, se muestran los tiempos de ejecución excluyendo el cálculo de la matriz de covarianzas. Los resultados demuestran que este algoritmo es muy adecuado para su implementación en FPGA dado que consigue un *speedup* por encima de 6 incluso para casos de imágenes con pocas bandas y píxeles. Además, debemos tener en cuenta que los procesadores que actualmente se utilizan para sistemas embarcados en satélites, ofrecen un menor rendimiento que el utilizado en los experimentos.

	Tiempo de ejecución		
	Pruebas 4 bandas 16 píxeles	**AVIRIS 32 bandas 614x512 píxeles**	**HYDICE 32 bandas 64x64 píxeles**
Implementación RX en C	0,272 milisegundos	1,533 segundos	19,736 milisegundos
Implementación RX propuesta	302 ciclos * 14.555 ns = 0,0044 milisegundos	15.092.010 ciclos * 15.466 ns = 0,234 segundos	198.954 ciclos * 15.466 ns = 3,08 milisegundos

Tabla 6.7: Comparación de las distintas implementaciones del algoritmo RX.

Debajo podemos encontrar las fórmulas en que nos hemos basados para obtener el tiempo ejecución (el número de ciclos necesarios):

$$T_{inversa} = [2 + (1 + 2 * (bandas - 1) + 7) * bandas] + [2 * bandas + 6] + 1$$

$$T_{mults\ \delta^{RX}} = pixels * [2 + (bandas + \log_2(bandas) + 2) + (\log_2(bandas) + 2)]$$

$$T_{total} = bandas + T_{inversa} + T_{mults\ \delta^{RX}} + 1$$

Capítulo 7

Conclusiones

El procesado a bordo de imágenes hiperespectrales de la superficie terrestre ha sido un objetivo muy perseguido en el campo de la teledetección. El número de aplicaciones que requieren una respuesta en tiempo real ha crecido exponencialmente en los últimos años. El diseño actual de sensores puede verse enormemente beneficiado con la incorporación de módulos especializados para el procesamiento, como las FPGAs, que pueden ser fácilmente empotradas en el sensor debido a su tamaño compacto.

En determinadas situaciones no es viable realizar un análisis completo de la imagen captada por el excesivo tiempo de ejecución que ello conllevaría, y resulta más eficaz aplicar técnicas de análisis consistentes en la detección de anomalías. En este trabajo se ha descrito una implementación en FPGA del algoritmo RX desarrollado por Reed y Yu, una de las aproximaciones más conocidas para realizar dicha detección.

Los resultados experimentales llevados a cabo en la FPGA Virtex-5 XC5VFX130T, demuestran que la implementación hardware propuesta obtiene resultados equivalentes a los que podemos encontrar en la literatura y puede superar de manera significativa (en términos de tiempo de cálculo) una versión de software equivalente. Las cualidades mencionadas del sistema reconfigurable propuesto hacen que sea muy adecuado para el procesamiento a bordo de imágenes hiperespectrales.

Bibliografía

[1] D. Landgrebe, "Hyperspectral image data analysis," Ieee Signal Processing Magazine, vol. 19, pp. 17-28, Jan 2002.

[2] C.-I. Chang, Hyperspectral Imaging: Techniques for Spectral Detection and Classification: New York, 2003.

[3] L. O. Jimenez and D. A. Landgrebe, "Supervised classification in high-dimensional space: Geometrical, statistical, and asymptotical properties of multivariate data," Ieee Transactions on Systems Man and Cybernetics Part C-Applications and Reviews, vol. 28, pp. 39-54, Feb 1998.

[4] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, et al., "Imaging spectroscopy and the Airborne Visible Infrared Imaging Spectrometer (AVIRIS)," Remote Sensing of Environment, vol. 65, pp. 227-248, Sep 1998.

[5] A. Plaza and C.-I. Chang, "Impact of initialization on design of endmember extraction algorithms," Ieee Transactions on Geoscience and Remote Sensing, vol. 44, pp. 3397-3407, Nov 2006.

[6] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "Spatial/spectral endmember extraction by multidimensional morphological operations," Ieee Transactions on Geoscience and Remote Sensing, vol. 40, pp. 2025-2041, Sep 2002.

[7] C. Gonzalez, J. J. Resano, D. Mozos, and A. Plaza, "Procesamiento a bordo de imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable," Universidad Complutense, Madrid, 2012.

[8] R. N. Clark, Spectroscopy of Rocks and Minerals, and Principles of Spectroscopy vol. 3: John Wiley and Sons, Inc, 1999.

[9] F. M. Lacar, M. M. Lewis, I. T. Grierson, and I. Ieee, "Use of hyperspectral imagery for mapping grape varieties in the Barossa Valley, South Australia," Igarss 2001: Scanning the Present and Resolving the Future, Vols 1-7, Proceedings, pp. 2875-2877, 2001 2001.

[10] J. G. Ferwerda, Charting the quality of forage: measuring and mapping the variation of chemical components in foliage with hyperspectral remote sensing: ITC Dissertation, 2005.

[11] A. K. Tilling, G. J. O'Leary, J. G. Ferwerda, S. D. Jones, G. J. Fitzgerald, D. Rodriguez, et al., "Remote sensing of nitrogen and water stress in wheat," ed: Elsevier B.V., 2007.

[12] J. A. F. Pierna, V. Baeten, A. M. Renier, R. P. Cogdill, and P. Dardenne, "Combination of support vector machines (SVM) and near-infrared (NIR) imaging spectroscopy for the detection of meat and bone meal (MBM) in compound feeds," Journal of Chemometrics, vol. 18, pp. 341-349, Jul-Aug 2004.

[13] H. Holma, A.-J. Mattila, T. Hyvarinen, and O. Weatherbee, "Advances in hyperspectral LWIR pushbroom imagers," in Conference on Next-Generation Spectroscopic Technologies IV, Orlando, FL, 2011.

[14] H. M. A. Werff, "Knowledge based remote sensing of complex objects: recognition of spectral and spatial patterns resulting from natural hydrocarbon seepages," Universiteit Utrecht, ITC Dissertation 131, 2006.

[15] M. F. Noomen, Hyperspectral Reflectance of Vegetation Affected by Underground Hydrocarbon Gas Seepage: ITC publication 145, 2007.

[16] M. Ambinder, "The secret team that killed bin Laden," in National Journal, ed, 2011.

[17] V. Farley, M. Charnberland, P. Lagueux, and A. Vallieres, "Chemical agent detection and identification with a hyperspectral Imaging infrared sensor - art. no. 66610L," in Conference on Imaging Spectrometry XII, San Diego, CA, 2007, pp. L6610-L6610.

[18] P. Tremblay, S. Savary, M. Rolland, A. Villemaire, M. Chamberland, V. Farley, et al., "Standoff gas identification and quantification from turbulent stack plumes with an imaging Fourier-transform spectrometer," in Conference on Advanced Environmental, Chemical, and Biological Sensing Technologies VII, Orlando, FL, 2010.

[19] J. Tuley, Soft Computing Reconfigures Designer Options. Embedded Systems, 1997.

[20] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," Proceedings of the Ieee, vol. 90, pp. 1201-1217, Jul 2002.

[21] M. Ahrens, A. Elgamal, D. Galbraith, J. Greene, S. Kaptanoglu, K. R. Dharmarajan, et al., "AN FPGA FAMILY OPTIMIZED FOR HIGH-DENSITIES AND REDUCED ROUTING DELAY," Proceedings of the Ieee 1990 Custom Integrated Circuits Conference, pp. 754-757, 1990 1990.

[22] H. C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, et al., "3RD-GENERATION ARCHITECTURE BOOSTS SPEED AND DENSITY OF FIELD-PROGRAMMABLE GATE ARRAYS," Proceedings of the Ieee 1990 Custom Integrated Circuits Conference, pp. 739-745, 1990 1990.

[23] Xilinx, "Using Block SelectRAM + Memory in Spartan II FPGAs," ed: Xilinx Application Notes, 2000.

[24] Xilinx. (2009). Available: http://www.xilinx.com/support/documentation/ip_documentation/ppc405_virtex4.pdf

[25] Z. Li, K. Compton, and S. Hauck, "Configuration Caching Management Techniques for Reconfigurable Computing," in IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00), 2000.

[26] J. Resano, J. A. Clemente, C. González, and D. Mozos, "HW implementation of an execution manager for reconfigurable systems," in The International Conference on Engineering of Reconfigurable Systems and Algorithms (ER-SA'07), 2007, pp. 71-78.

[27] J. Resano, J. Antonio Clemente, C. Gonzalez, D. Mozos, and F. Catthoor, "Efficiently Scheduling Runtime Reconfigurations," Acm Transactions on Design Automation of Electronic Systems, vol. 13, Sep 2008.

[28] T. Anderson, "The reality of using cores as virtual components," Electronic Engineering, vol. 70, pp. 27-+, Jul-Aug 1998.

[29] S. Trimberger, D. Carberry, A. Johnson, J. Wong, and S. O. C. Ieee Comp, "A time-multiplexed FPGA," 5th Annual Ieee Symposium on Field-Programmable Custom Computing Machines, pp. 22-28, 1997 1997.

[30] A. DeHon, "Reconfigurable Architectures for General Purpose Computing," Artificial Intelligence Laboratory, Massachusetts Institute of Technology1996.

[31] J. A. Clemente, J. Resano, C. González, and D. Mozos, "A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ed, 2010, pp. 1-14.

[32] Xilinx. (2003). Virtex 2.5V Field Programmable Gate Array. Available: http://www.xilinx.com

[33] Verilog. (2008). IEEE Standard Verilog Hardware Description Language. Available: http://www.verilog.com/IEEEVerilog.html

[34] VHDL. IEEE VHDL Analysis and Standarization Group. Available: http://www.vhdl.org/vasg, 2006

[35] S. Bhunia, M. Tabib-Azar, D. Saab, and Ieee, "Ultralow-power reconfigurable computing with complementary nano-electromechanical carbon nanotube switches," in 12th Asia and South Pacific Design Automation Conference, Yokohama, JAPAN, 2007, pp. 86-91.

[36] R. O. Reynolds, P. H. Smith, L. S. Bell, and H. U. Keller, "The design of Mars lander cameras for Mars Pathfinder, Mars Surveyor '98 and Mars Surveyor '01," Ieee Transactions on Instrumentation and Measurement, vol. 50, pp. 63-71, Feb 2001.

[37] M. Kifle, M. Andro, Q. K. Tran, G. Fujikawa, and P. P. Chu, "Toward a dynamically reconfigurable computing and communication system for small spacecraft," in 21st International Communication Satellite System Conference & Exhibit (ICSSC '03) 2003.

[38] Triscend, "Configurable Processors: An Emerging Solution for Embedded System Design," ed: A White Paper, 1998.

[39] Altera. Available: http://www.altera.co.jp/literature/sg/sg_ip.pdf. 2005

[40] J. Tabero, H. Mecha, J. Septién, S. Román, and D. Mozos, "A Vertex-List Approach to 2D Hw Multitasking Management in RTR FPGAs," in DCIS, 2003, pp. 545-550.

[41] S. Roman, J. Septien, H. Mecha, and D. Mozos, "Constant complexity management of 2D HW multitasking in run-time

reconfigurable FPGAs," Reconfigurable Computing: Architectures and Applications, vol. 3985, pp. 187-192, 2006 2006.

[42] J. A. Clemente, C. Gonzalez, J. Resano, and D. Mozos, "A task graph execution manager for reconfigurable multi-tasking systems," Microprocessors and Microsystems, vol. 34, pp. 73-83, Mar-Jun 2010.

[43] J. T. Thomson. Rad Hard FPGAs. Available: http://esl.eng.ohiostate.edu/~rstheory/iip/RadHardFPGA.doc

[44] I. S. Reed and X. L. Yu, "Adaptive multiple-band CFAR detection of an optical pattern with unknown spectral distribution," Ieee Transactions on Acoustics Speech and Signal Processing, vol. 38, pp. 1760-1770, Oct 1990.

[45] J. A. Richards and J. X, Remote sensing digital image analysis: an introduction. Berlin: Springer, 2006.

[46] J. M. Molero, A. Paz, E. M. Garzon, J. A. Martinez, A. Plaza, and I. Garcia, "Fast anomaly detection in hyperspectral images with RX method on heterogeneous clusters," Journal of Supercomputing, vol. 58, pp. 411-419, Dec 2011.

[47] J. M. Molero, E. M. Garzon, I. Garcia, and A. Plaza, "Parallel Implementation of RX Anomaly Detection on Multi-Core Processors: Impact of Data Partitioning Strategies," in High-Performance Computing in Remote Sensing. vol. 8183, B. Huang and A. J. Plaza, Eds., ed, 2011.

[48] Xilinx. Available online: http://www.xilinx.com/publications/prod_mktg/virtex5qv-product-table.pdf.

[49] "Escena AVIRIS WTC," Available: http://speclab.cr.usgs.gov/wtc/

[50] "Mapa termal del USGS," Available: http://pubs.usgs.gov/of/2001/ofr-01-0429/hotspot.key.tgif.gif

[51] A. Paz, J. M. Molero, E. M. Garzon, J. A. Martinez, and A. Plaza, "A New Parallel Implementation of the RX Algorithm for Anomaly Detection in Hyperspectral Images," in 9th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'10), Almería, Spain, 2010.

[52] J. M. Molero, E. M. Garzon, I. Garcia and A. Plaza. Analysis and Optimizations of Global and Local Versions of the RX Algorithm for Anomaly Detection in Hyperspectral Data. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, accepted for publication (pdf), 2013 [JCR(2011)=1.489].

Acrónimos

FPGA: Field Programmable Gate Array.

Algoritmo RX: Algoritmo de Reed y Yu.

VHDL: Very high speed integrated circuit (VHSIC) Hardware Description Language.

HDL: Hardware Description Language.

AVIRIS: Airborne Visible/Infrared Imaging Spectrometer.

WTC: World Trade Center.

HYDICE: Hyperspectral Digital Imagery Collection Experiment.

EnMAP: Environmental Mapping and Analysis Program (German Aerospace Center).

NASA: National Aeronautics and Space Administration.

PCA: Principal Component Analysis.

ENVI: Environment for Visualizing Images.

ASI: Agenzia Spaziale Italiana.

ASIC: Application Specific Integrated Circuits.

LB: Logic Blocks.

IOB: Input-Output-Blocks.

RAM: Random Access Memory.

SRAM: Static Random Access Memory.

IEEE: Institute of Electrical and Electronics Engineers.

GPU: graphics processing unit.

Xilinx ISE: Xilinx Integrate Software Enviroment.

PF: Punto Flotante.

IP-core: Intellectual Property core.

USGS: U.S. Geological Survey.